CS477 - Formal Software Development Methods

Verification of Concurrent Programs

Grigore Roşu

(slides from José Meseguer)

Department of Computer Science
University of Illinois at Urbana-Champaign

Verification of Concurrent Programs

We will begin considering the topic of verification of concurrent programs. As for sequential programs, we will consider first the case of *declarative* concurrent programs. Later in the course we will also consider *imperative* concurrent programs.

So the first question is, what is a *suitable computational logic* to write concurrent programs in a declarative style? This is of course an *open-ended* question, in that a variety of answers are possible at present, and new answers may be proposed in the future.

Verification of Concurrent Programs (II)

In this course, we will use *rewriting logic* as a specific computational logic that is indeed suitable for concurrent programming.

This is in full harmony with our use of equational logic for what, rather than sequential, we could better call *deterministic* declarative programming. In fact, rewriting logic *generalizes* equational logic in a natural way.

Rewrite Theories: Preliminary Definition

We give a first, already quite general, definition of rewrite theories. We will further generalize this notion later.

A *rewrite theory* $\mathcal{R}$ is a triple $\mathcal{R} = (\Sigma, E, R)$, with:

- (Σ, E) a membership equational theory, and
- R a set of *labeled rewrite rules* of the form $l : t \longrightarrow t' \Leftarrow cond$, with l a label, $t, t' \in T_\Sigma(X)_k$ for some kind k, and $cond$ a *condition* (involving the same variables X) as explained below.

Conditional Rewrite Rules

The most general form of a conditional rewrite rule is:

$$l : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k),$$

that is, in general, the condition is a conjunction of *equations*, *memberships*, and *rewrites*, where the variables in all the Σ-terms $t, t', u_i, u'_i, v_j, w_k, w'_k$ are contained in a common set X. There is *no* requirement that $vars(t) = X$, and *no* assumptions of confluence or termination. The rule is called *unconditional* if the condition is empty.

Maude System Modules

In Maude, rewrite theories are specified in *system modules*.

The same way that a functional module has essentially the form, `fmod` (Σ, E) `endfm`, with (Σ, E) a membership equational logic theory, a system module has essentially the form, `mod` (Σ, E, R) `endm`, with (Σ, E, R) a rewrite theory.

We will illustrate the syntax details in examples. In particular, a conditional rewrite rule of the form, $l : t \longrightarrow t' \Leftarrow cond$ is specified in Maude with syntax,

`crl [`l`] :` t `=>` t' `if` $cond$ `.`

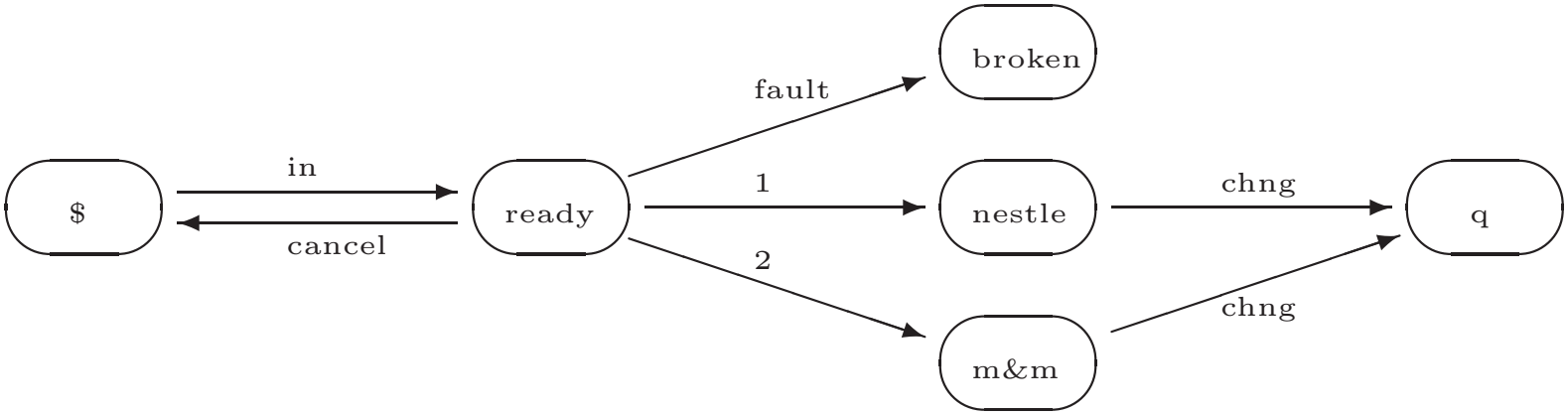
Some Rewriting Logic Examples

To motivate rewriting logic as a formalism to specify and program concurrent systems, we will show how it can be used to naturally specify three important classes of systems, namely:

- automata, also called *labeled transition systems*,
- *Petri nets*, one of the simplest concurrency models, and
- *object-oriented* concurrent systems.

Concurrency vs. Nondeterminism: Automata

We can motivate concurrency by its absence. The point is that we can have systems that are *nondeterministic*, but are not concurrent. Consider the following faulty automaton to buy candy:



Concurrency vs. Nondeterminism: Automata (II)

Although in the standard terminology this would be called a *deterministic* automaton (because each labeled transition from each state leads to a single next state) in reality it is still *nondeterministic*, in the sense that its computations *are not confluent*, and therefore *completely different outcomes* are possible.

For example, from the `ready` state the transitions `fault` and `1` lead to completely different states that can never be reconciled in a common subsequent state.

Concurrency vs. Nondeterminism: Automata (III)

So, the automaton is in this sense nondeterminisitc, yet it is *strictly sequential*, in the sense that, although at each state the automaton may be able to take several transitions, it can only take *one transition at a time*.

Since the intuitive notion of concurrency is that *several transitions can happen simultaneously*, we can conclude by saying the our automaton, although it exhibits a form of nondeterminism, *has no concurrency* whatsoever.

Automata as Rewrite Theories

We can specify such an automaton as a system module,

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State .
  rl [in] : $ => ready .
  rl [cancel] : ready => $ .
  rl [1] : ready => nestle .
  rl [2] : ready => m&m .
  rl [fault] : ready => broken .
  rl [chng] : nestle => q .
  rl [chng] : m&m => q .
endm
```

Rewite Rules as Transitions

Note that *rewrite rules* do *not* have an equational interpretation. They are *not* understood as equations, but as *transitions*, that in general *cannot be reversed*.

This is why, in a rewite theory (Σ, E, R) the equations in E are *totally different* from the rules R, since equations and rules have a *totally different semantics*.

However, *operationally* Maude will assume that the equations in E are confluent, terminating, and sort decreasing modulo some $A \subseteq E$, and will compute with such equations and also with the rules in R by rewriting, yet distinguishing *equation simplification* (the `reduce` command) from *rewriting with rules* (the `rewrite` command).

The `rewrite` Command

Maude can execute rewrite theories with the **`rewrite`** command (can be abbreviated to **`rew`**). For example,

```
Maude> rew $ .
rewrite in CANDY-AUTOMATON : $ .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result State: q
```

The **`rewrite`** command applies the rules in a *fair* way (all rules are given a chance) until termination, and gives one result.

The `rewrite` Command (II)

In this example, fairness saves us from nontermination, but in general we can esily have nonterminating computations.

For this reason the **`rewrite`** command can be given a numeric argument stating the *maximum number of rewrite steps*. For example,

The rewrite Command (III)

```
Maude> set trace on .
Maude> rew [3] $ .
rewrite [3] in CANDY-AUTOMATON : $ .
********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
********** rule
rl [cancel]: ready => $ .
empty substitution
ready ---> $
********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result State: ready
```

The `search` Command

Of course, since we are in a nondeterministic situation, the `rewrite` command gives us *one possible behavior* among many.

To systematically explore *all behaviors* from an initial state we can use the `search` command, which takes two terms: a ground term which is our initial state, and a term, possibly with variables, which describes our desired target state.

Maude then does a *breadth first search* to try to reach the desired target state. For example, to find the terminating states from the `$` state we can give the command (where the "`!`" in `=>!` specifies that the target state must be a terminating state),

The search Command (II)

```
Maude> search $ =>! X:State .
search in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken

Solution 2 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

We can then inspect the search graph by giving the command,

The search Command (III)

```
Maude> show search graph .
state 0, State: $
arc 0 ===> state 1 (rl [in]: $ => ready .)

state 1, State: ready
arc 0 ===> state 0 (rl [cancel]: ready => $ .)
arc 1 ===> state 2 (rl [1]: ready => nestle .)
arc 2 ===> state 3 (rl [2]: ready => m&m .)
arc 3 ===> state 4 (rl [fault]: ready => broken .)

state 2, State: nestle
arc 0 ===> state 5 (rl [chng]: nestle => q .)

state 3, State: m&m
arc 0 ===> state 5 (rl [chng]: m&m => q .)

state 4, State: broken
state 5, State: q
```

The search Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 5) by giving the command,

```
Maude> show path 5 .
state 0, State: $
===[ rl [in]: $ => ready . ]===>
state 1, State: ready
===[ rl [1]: ready => nestle . ]===>
state 2, State: nestle
===[ rl [chng]: nestle => q . ]===>
state 5, State: q
```

The `search` Command (V)

Similarly, we can search for target terms reachable by *one* rewrite step, *one or more*, or *zero or more* steps by typing (respectively):

- `search` t `=>1` t' .
- `search` t `=>+` t' .
- `search` t `=>*` t' .

The search Command (VI)

Furthermore, we can restrict any of those searches by giving an *equational condition* on the target term. For example, all terminating states reachable from `$` other than `broken` can be found by the command,

```
Maude> search $ =>! X:State such that X:State =/= broken .
search in CANDY-AUTOMATON : $ =>! X:State
such that X:State =/= broken = true .

Solution 1 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

The `search` Command (VII)

Of course, in general there can be an *infinite* number of solutions to a given search. Therefore, a search can be restricted by giving as an extra parameter in brackets the number of solutions (i.e., target terms that are instances of the pattern and satisfy the condition) we want:

```
search [1] in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken
```

Labelled Transition Systems

Our `CANDY-AUTOMATON` example is just a special instance of a general concept, namely, that of *automaton*, also called a *labeled transition system* (LTS) by which we mean a triple: $A = (A, L, T)$ with:

- A is a set, called the set of *states*,
- L is a set called the set of *labels*, and
- $T \subseteq A \times L \times A$ is called the set of *labeled transitions*.

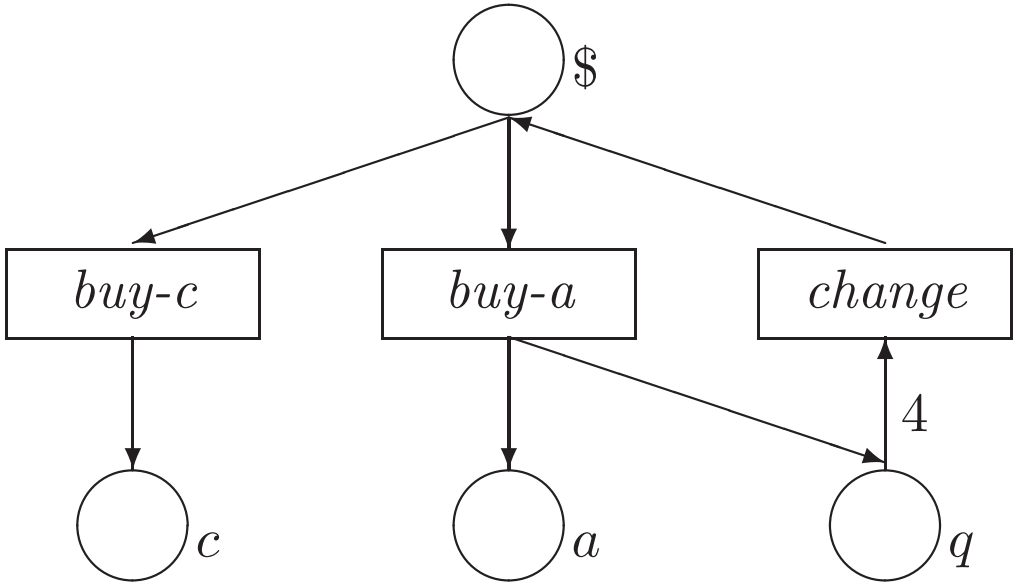
LTS's as Rewrite Theories

Note that we have associated to our candy automaton a rewrite theory (system module) `CANDY-AUTOMATON`.

This is of course just an instance of *a general transformation*, that assign to a LTS A a rewrite theory $R(A)$ with a single sort A, constants $x \in A$, and for each $(x, l, y) \in T$ a rewrite rule $l : x \longrightarrow y$.

Petri Nets

So far so good, but we have not yet seen any concurrency. The simplest concurrent system examples are probably the *concurrent automata* called *Petri nets*. Consider for example the picture,



Petri Nets (II)

The previous picture represents a concurrent machine to buy cakes and apples; a cake costs a dollar and an apple three quarters.

Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is *concurrent*, because we can *push several buttons* at once, provided enough resources exist in the corresponding slots, which are called *places*

Petri Nets (III)

For example, if we have one dollar in the \$ place, and four quarters in the q place, we can *simultaneously* push the *buy-a* and *change* buttons, and the machine returns, also simultaneously, one dollar in \$, one apple in a, and one quarter in q.

That is, we can achieve the *concurrent computation*,

$$\textit{buy-a}\ \ \textit{change}\ :\ \$\ q\ q\ q\ q \longrightarrow a\ q\ \$.$$

Petri Nets (IV)

This has a straightforward expression as a rewrite theory (system module) as follows:

```
mod PETRI-MACHINE is
  sort Marking .
  ops null $ c a q : -> Marking .
  op _ _ : Marking Marking -> Marking [assoc comm id: null] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [chng] : q q q q => $ .
endm
```

Petri Nets (V)

That is, we view the *distributed state* of the system as a *multiset of places*, called a *marking*, with identity for multiset union the empty multiset `null`.

We then view a *transition* as a *rewite rule* from one (pre-)marking to another (post-)marking.

Petri Nets (VI)

The rewrite rule can be applied *modulo associativity, commutativity and identity* to the distributed state iff its pre-marking is a submultiset of that state.

Furthermore, if the distributed state contains the *union* of several such presets, then *several transitions* can fire *concurrently*.

For example, from `$ $ $` we can get in *one concurrent step* to `c c a q` by pushing twice (concurrently!) the `buy-c` button and once the `buy-a` button.

Petri Nets (VII)

We can of course ask and get answers to questions about the behaviors possible in this system. For example, if I have a dollar and three quarters, can I get a cake and an apple?

```
Maude> search $ q q q =>+ c a M:Marking .
search in PETRI-MACHINE : $ q q q =>+ c a M:Marking .

Solution 1 (state 4)
states: 5 in 0ms cpu (0ms real)
M:Marking --> null
```

we can also interrogate the search graph,

Petri Nets (VIII)

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ===> state 1 (rl [buy-c]: $ => c .)
arc 1 ===> state 2 (rl [buy-a]: $ => a q .)

state 1, Marking: c q q q

state 2, Marking: a q q q q
arc 0 ===> state 3 (rl [chng]: q q q q => $ .)

state 3, Marking: $ a
arc 0 ===> state 4 (rl [buy-c]: $ => c .)
arc 1 ===> state 5 (rl [buy-a]: $ => a q .)

state 4, Marking: c a

state 5, Marking: a a q
```

Petri Nets (IX)

```
Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl [buy-a]: $ => a q . ]===>
state 2, Marking: a q q q q
===[ rl [chng]: q q q q => $ . ]===>
state 3, Marking: $ a
===[ rl [buy-c]: $ => c . ]===>
state 4, Marking: c a
```

What is Concurrency?

Why was concurrency *impossible* in our `CANDY-AUTOMATON` example, but possible in our little `PETRI-MACHINE` example?

The problem with `CANDY-AUTOMATON`, and with any LTS having unstructured states, is that its states are *atomic*, and, having no smaller pieces, *cannot be distributed*.

By contrast, a Petri net marking *is made out of smaller pieces*, namely its constituent places, and therefore *can be distributed*, so that several transitions can happen simultaneously.

What is Concurrency? (II)

Then what, is concurrency about multisets?

Not necessarily; this is the very common fallacy of *taking the part for the whole*; for example, "Logic Programming = Prolog," or "Concurrency = Petri Nets".

A more fair and open-minded answer is to give the rewriting logic motto:

Concurrent Structure = Algebraic Structure.

What is Concurrency? (III)

That is, *any algebraic structure* in the set of states, other than atomic constants, even a single unary operator, will open the possibility for the states to be *distributed*, and therefore for transitions being concurrent.

Of course that potential for concurrency may be frustrated by the specific transitions of a system *forcing a sequential execution*, but the potential is there if we use other transitions.

In summary, there are *as many possible styles of concurrent systems* as there are *signatures* Σ and equations E. For example: multiset concurrency, tree concurrency, string concurrency, and many, many other possibilities.

Petri Nets in General

I give the Meseguer-Montanari "Petri nets are monoids" definition, instead than the usual, but less enlightening, multigraph definition.

A *place-transition* Petri net N consists of:

- a set P of *places*; we then call *markings* to the elements in the free commutative monoid $M(P)$ of finite multisets of P.
- a labeled transition system $N = (M(P), L, T)$.

Petri Nets in General (II)

The general transformation associating a rewrite theory $R(N)$ to each Petri net N is then obvious. $R(N)$ has:

- a single sort, named, say $M(P)$, or just *Marking*, with constants the elements of P and a *null* constant.
- a binary operator
  $\_\ \_ : Marking\ Marking \longrightarrow Marking\ [assoc\ comm\ id :\ null]$
- for each $(m, l, m') \in T$ a rewrite rule $l : m \longrightarrow m'$.

Petri Net Computations

The computations of a net N are not just paths, since we can now take several concurrent steps at once. They are generated as follows:

- **Reflexivity**.

$$\frac{m \in M(P)}{m \xrightarrow{m} m}$$

- **Basic Transition**.

$$\frac{(m, l, m') \in T}{m \xrightarrow{l} m'}$$

- **Congruence**.

$$\frac{m \xrightarrow{\alpha} m' \qquad u \xrightarrow{\beta} u'}{m\ u \xrightarrow{\alpha\ \beta} m'\ u'}$$

Petri Net Computations (II)

- **Transitivity**.

$$\frac{m \xrightarrow{\alpha} u \qquad u \xrightarrow{\beta} v}{m \xrightarrow{\alpha;\beta} v}$$

We will see later that, when we view Petri nets as rewite theories, the above inference system generating all Petri net computations of a net N *coincides* with the *specialization* of the general inference system of rewriting logic to the rewrite theory $R(N)$.

This illustrates a general point, namely, that rewriting logic is a very expressive *semantic framework*, in which many different concurrency models can be naturally specified.

Concurrent Objects in Rewriting Logic

Rewriting logic can model very naturally many different kinds of concurrent systems. We have, for example, seen that Petri nets can be naturally formalized as rewrite theories. The same is true for many other models of concurrency such as CCS, the π-calculus, dataflow, real-time models, and so on.

One of the most useful and important classes of concurrent systems is that of *concurrent object systems*, made out of *concurrent objects*, which encapsulate their own local state and can *interact* with other objects in a variety of ways, including both *synchronous interaction*, and *asynchronous communication by message passing*.

Concurrent Objects in Rewriting Logic (II)

It is of course possible to *represent* a concurrent object system as a rewrite theory with somewhat different modeling styles and adopting different *notational conventions*.

What follows is a particular style of representation that has proved useful and expressive in practice, and that is supported by Full Maude's *object-oriented modules*.

It is also possible to define object-oriented modules in Core Maude using the `conf` attribute to specify an associative commutative multiset union operators as a constructor of configurations of objects and messages; the `frewrite` command then ensures object and message fair executions (see the Maude 2.0 manual).

Concurrent Objects in Rewriting Logic (III)

To model a concurrent object system as a rewrite theory, we have to explain two things:

- how the *distributed states* of such a system are equationally axiomatized and modeled by the initial algebra of an equational theory (Σ, E), and
- how the *concurrent interactions* between objects are *axiomatized by rewrite rules*.

We first explain how the distributed states are equationally axiomatized.

Configurations

Let us consider the key state-building operations in Σ and the equations E axiomatizing the distributed states of concurrent object systems. The concurrent state of an object-oriented system, often called a *configuration*, has typically the structure of a *multiset* made up of *objects* and *messages*.

Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax (i.e. juxtaposition) as,

Configurations (II)

The operator `_ _` is declared to satisfy the structural laws of *associativity and commutativity* and to have *identity* `null`. Objects and messages are singleton multiset configurations, and belong to subsorts

$$\textbf{Object Msg} < \textbf{Conf},$$

so that more complex configurations are generated out of them by multiset union.

Configurations (III)

An *object* in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the *object's name* or identifier, C is its *class*, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*.

The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator _ , _ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

Configurations (IV)

The value of each attribute shouldn't be arbitrary: it should have an appropriate *sort*, dictated by the nature of the attribute. Therefore, in Full Maude *object classes* can be declared in *class declarations* of the form,

$$\texttt{class}\ C \mid a_1 : s_1, \ldots, a_n : s_n\ .$$

where C is the class name, and s_i is the sort required for attribute a_i.

We can illustrate such class declarations by considering three classes of objects, `Buffer`, `Sender`, and `Receiver`.

Configurations (IV)

A *buffer* stores a list of integers in its `q` attribute. Lists of integers are built using an associative list concatenation operator, `_._` with identity `nil`, and integers are regarded as lists of length one. The name of the object reading from the buffer is stored in its `reader` attribute; such names belong to a sort `Oid` of *object identifiers*. Therefore, the class declaration for buffers is,

```
class Buffer | q : IntList, reader: Oid .
```

The *sender* and *receiver* objects store an integer in a `cell` attribute that can also be empty (`mt`) and have also a counter (`cnt`) attribute. The sender stores also the name of the receiver in an additional attribute.

Configurations (V)

The counter attribute is used to ensure that messages are received by the receiver in the same order as they are sent by the sender, even though communication between the two parties is asynchronous.

Each time the sender gets a new value from the buffer, it increments its counter. It later uses the current value of the counter to tag the message sent with that value to the receiver.

The receiver only accepts a message whose tag is its current counter. It then increments its counter indicating that it is ready for the next message.

Configurations (VI)

The class declarations are:

```
class Sender | cell: Int?, cnt: Int, receiver: Oid .
class Receiver | cell: Int?, cnt: Int .
```

where `Int?` is a supersort of `Int` having a new constant `mt`.

In Full Maude one can also give *subclass declarations*, with `subclass` syntax (similar to that of `subsort`) so that all the attributes and rewrite rules of a superclass are *inherited* by a subclass, which can have additional attributes and rules of its own.

Configurations (VII)

The *messages* sent by a sender object have the form,

```
(to Z : E from (Y,N))
```

where `Z` is the name of the receiver, `E` is the number sent, `Y` is the name of the sender, and `N` is the value of its counter at the time of the sending.

The syntax of messages is user-definable; it can be declared in Full Maude by message operator declarations. In our example by:

```
msg (to _ : _ from (_,_)) : Oid Int Oid Int -> Msg .
```

Object Rewrite Rules

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind.

In general, the rewrite rules in R describing the dynamics of an object-oriented system can have the form,

Object Rewrite Rules (II)

$$
\begin{aligned}
r: \quad & M_1 \ldots M_n \, \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\
\longrightarrow \; & \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
& \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& M'_1 \ldots M'_q \\
& \mathit{if}\ C
\end{aligned}
$$

where r is the label, the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition.

Object Rewrite Rules (III)

That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created.

If two or more objects appear in the lefthand side, we call the rule *synchronous*, because it forces those objects to jointly participate in the transition. If there is only one object and at most one message in the lefthand side, we call the rule *asynchronous*.

Object Rewrite Rules (IV)

Three typical rewrite rules involving objects in the `Buffer`, `Sender`, and `Receiver` classes are,

```
rl [read] : < X : Buffer | q: L . E, reader: Y >
                < Y : Sender | cell: mt, cnt: N >
             => < X : Buffer | q: L, reader: Y >
                < Y : Sender | cell: E, cnt: N + 1 >

rl [send] : < Y : Sender | cell: E, cnt: N, receiver: Z >
   => < Y : Sender | cell: mt, cnt: N > (to Z : E from (Y,N))

rl [receive] : < Z : Receiver | cell: mt, cnt: N >
                 (to Z : E from (Y,N))
                => < Z : Receiver | cell: E, cnt: N + 1 >
```

where `E` and `N` range over `Int`, `L` over `IntList`, `X`, `Y`, `Z` over `Oid`,

and `L . E` is a list with last element `E`.

Object Rewrite Rules (V)

Notice that the `read` rule is synchronous and the `send` and `receive` rules asynchronous.

Of course, these rules are applied *modulo* the associativity and commutativity of the multiset union operator, and therefore allow both object synchronization and message sending and receiving events anywhere in the configuration, regardless of the position of the objects and messages.

We can then consider the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizing the object system with these three object classes, with R the three rules above (and perhaps other rules, such as one for the receiver to write its contents into another buffer object, that are omitted).

Rewrite Theories in General

It is clear that *rewriting logic has the underlying equational logic as a parameter*: the more general the equational logic, the more general the resulting rewite theories. For example, we have seen that for membership equational logic rules can have the general form,

$$l : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w_k').$$

It has also become increasingly clear that *frozen operators*, that restrict the rewrites allowed below them, are also very useful in practice.

Rewrite Theories in General (II)

We can illustrate frozen operators with the following nondeterministic choice example (in Maude syntax):

```
mod CHOICE is
  protecting INT .
  sorts Elt MSet .
  subsorts Elt < MSet .
  ops a b c d e f g : -> Elt .
  op __ : MSet MSet -> MSet [assoc comm] .
  op card : MSet -> Int [frozen] .
  eq card(X:Elt) = 1 .
  eq card(X:Elt M:MSet) = 1 + card(M:MSet) .
  rl [choice] : X:MSet Y:MSet => Y:MSet .
endm
```

Rewrite Theories in General (III)

It does not make much sense to rewrite below the cardinality function `card`, because then the multiset whose cardinality we wish to determine becomes a *moving target*.

If `card` had not been declared `frozen`, then the rewrites, `a b c` $\longrightarrow$ `b c` $\longrightarrow$ `c` would induce rewrites, $3 \longrightarrow 2 \longrightarrow 1$, which seems bizarre.

The point is that we think of the kind `[MSet]` as the *state kind* in this example, whereas `[Int]` is the *data kind*. By declaring `card` frozen, we restrict rewrites to the state kind, where they belong.

Rewrite Theories in General (IV)

This leads to the following general definition of a rewrite theory on membership equational logic:

A *rewrite theory* is a 4-tuple, $\mathcal{R} = (\Sigma, E, \phi, R)$, where:

- (Σ, E) is a membership equational theory, with, say, kinds K, sorts S, and operations Σ
- $\phi : \Sigma \longrightarrow \mathcal{P}_{fin}(\mathbb{N})$ is a $K^* \times K$-indexed family of functions assigning to each $f : k_1 \ldots k_n \longrightarrow k$ in Σ the finite set $\phi(f) \subseteq \{1, \ldots, n\}$ of its *frozen argument positions*
- R is a set of (universally quantified) labeled conditional rewrite rules of the form (with t, t' and the w_k, w'_k pairs of terms of

same kind)

$$l : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w_k').$$

Rewrite Theories in General (V)

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, and given a Σ-term $t \in T_\Sigma(X)$, we call a variable $x \in vars(t)$ *frozen* in t iff there is a nonvariable position $\alpha \in \mathbb{N}^*$ such that $t/\alpha = f(u_1, \ldots, u_i, \ldots, u_n)$, with $i \in \phi(f)$, and $x \in vars(u_i)$. Otherwise, we call $x \in X$ *unfrozen*.

Similarly, given Σ-terms $t, t' \in T_\Sigma(X)$, we call a variable $x \in X$ *unfrozen* in t and t' iff it is unfrozen in both t and t'.

Rewriting Logic in General

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, the sentences that it proves are universally quantified rewrites of the form, $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_{\Sigma,E}(X)_k$, for some kind k, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$

- **Equality**.
$$\frac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X)u = u' \quad E \vdash (\forall X)v = v'}{(\forall X)\ u' \longrightarrow v'}$$

- **Congruence**. For each $f : k_1 \ldots k_n \longrightarrow k$ in Σ, with $\{1, \ldots, n\} - \phi(f) = \{j_1, \ldots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \ldots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \longrightarrow f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)}$$

- **Replacement**. For each finite substitution $\theta : X \longrightarrow T_\Sigma(Y)$, with, say, $X = \{x_1, \ldots, x_n\}$, and $\theta(x_l) = p_l$, $1 \leq l \leq n$, and for each rule in R of the form,

$$l : (\forall X)\, t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w_k')$$

with $Z = \{x_{j_1}, \ldots, x_{j_m}\}$, the set of unfrozen variables in t and t', then,

$$(\bigwedge_r (\forall Y)\, p_{j_r} \longrightarrow p'_{j_r})$$

$$\frac{(\bigwedge_i (\forall Y)\, \theta(u_i) = \theta(u_i')) \wedge (\bigwedge_j (\forall Y)\, \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\, \theta(w_k) \longrightarrow \theta(w_k')}{(\forall Y)\, \theta(t) \longrightarrow \theta'(t')}$$

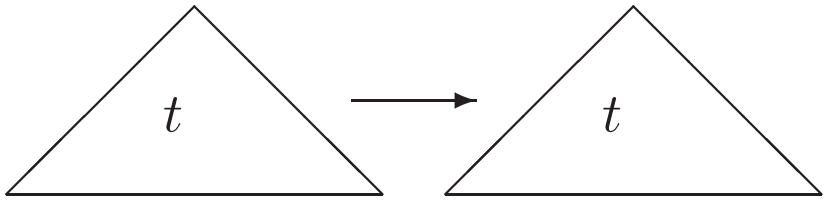
where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

- **Transitivity**

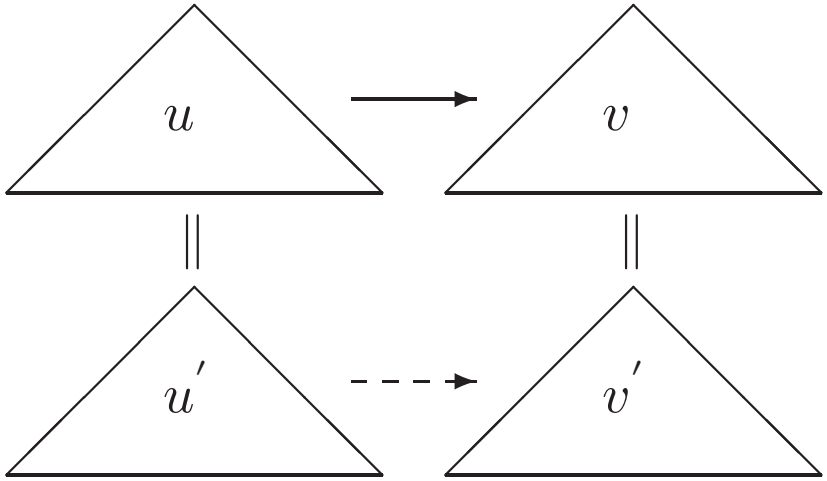
$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

Rewriting Logic in Pictures

Reflexivity

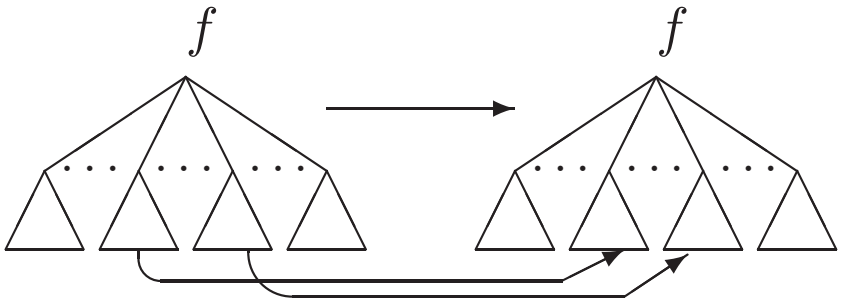


Equality

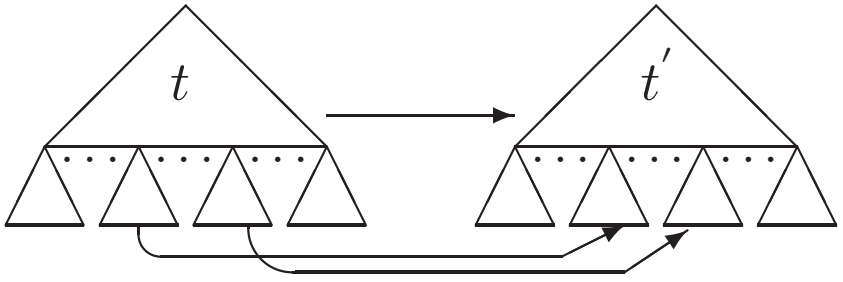


Rewriting Logic in Pictures (II)

Congruence

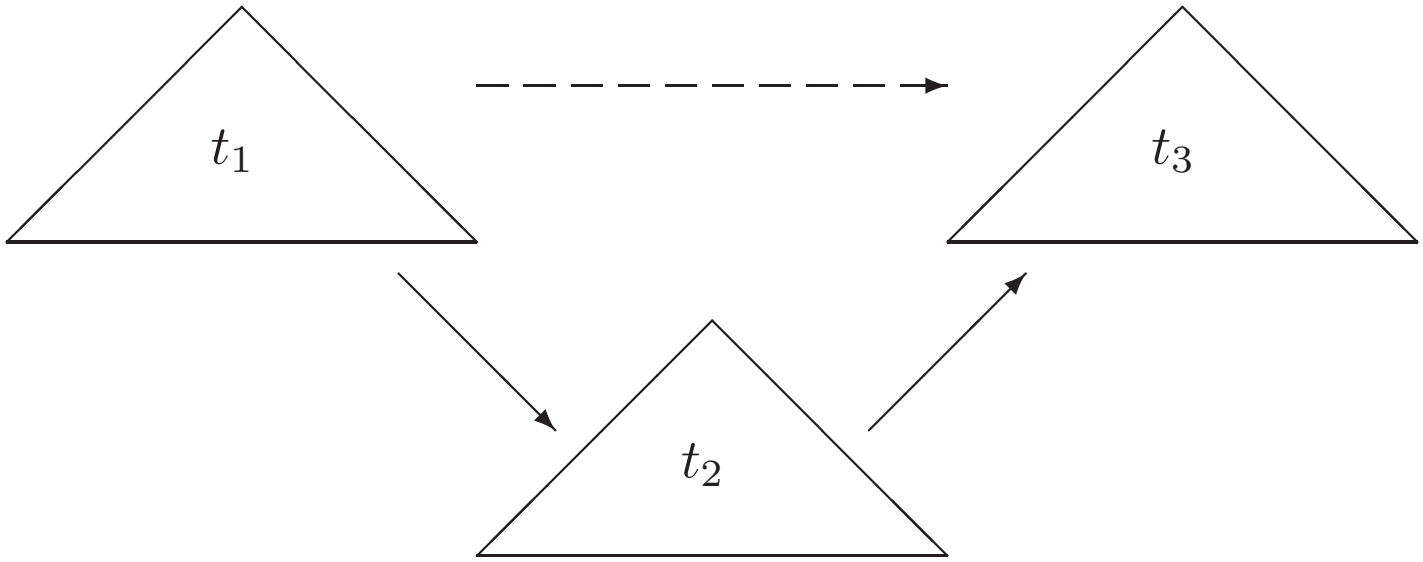


Replacement



Rewriting Logic in Pictures (III)

Transitivity



Computational Meaning of the Inference Rules

Rewriting logic is a *computational logic* to specify concurrent systems. Its inference system allows us to infer *all* the possible finitary concurrent computations of a system specified as a rewrite theory $\mathcal{R}$ as follows:

- **Reflexivity** is just the possibility of having *idle* transitions
- **Equality** means that states are equal *modulo* E
- **Congruence** is a general form of *sideways parallelism*
- **Replacement** combines an *atomic transition* at the top using a rule with *nested concurrency* in the substitution
- **Transitivity** is *sequential composition*.

Logical and Computational Readings

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has two closely related, yet different, *readings*, one computational, and another logical.

Computationally, a rewrite theory specifies a *concurrent system*, whose set of *states* is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies a parameterized family of *concurrent transitions* in the system.

Logically, a rewrite theory specifies a *logic*, whose set of *formulas* is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies an *inference rule* in the logic.

Logical and Computational Readings (II)

For example, the *logic of implication* is:

```
mod MINIMALR is sorts SentConstant Formula Configuration .
subsorts SentConstant < Formula < Configuration .
op _⟶_ : Formula Formula -> Formula .
op empty : -> Configuration .
op __ : Configuration Configuration -> Configuration [assoc comm id: empt
vars A B C : Formula .
rl [ax.K] :        empty
          => ---------------
              A → (B → A) .
rl [ax.S] :                          empty
          => ---------------------------------------
            (A → B) → ((A →(B → C)) → (A → C)) .
rl [mp] :      (A → B)   A
```

```
=> ------------------
```

$$B \quad .$$

```
endm
```

Logical and Computational Readings (III)

The computational and logical readings are not *mutually exclusive*. Rather, the *same theory* can be regarded computationally, or logically, or *both*!, depending on one's point of view.

For example, logically, our `PETRI-MACHINE` example is a *linear logic theory* in disguise. It is just a matter of a slight change of syntax, replacing the empty syntax, _ _ by that of linear logic's *multiplicative conjunction* operator $\_ \otimes \_$.

Logical and Computational Readings (IV)

The operator $\_ \otimes \_$ can be viewed as a form or *resource-conscious* non-idempotent conjunction. Then, the state $a \otimes q \otimes q$ corresponds to having an apple *and* a quarter *and* a quarter, which is a strictly better situation than having an apple *and* a quarter (non-idempotence of $\otimes$).

Then, in order to get the tensor theory corresponding to `PETRI-MACHINE`, it is enough to change the arrows into turnstiles, getting the following axioms:

$$\begin{array}{ll} \textit{buy-c}: & \$ \vdash c \\ \textit{buy-a}: & \$ \vdash a \otimes q \\ \textit{change}: & q \otimes q \otimes q \otimes q \vdash \$ \end{array}$$

Logical and Computational Readings (V)

The point is that we have the following equivalences between these two readings:

State	$\longleftrightarrow$	*Term*	$\longleftrightarrow$	*Formula*
Computation	$\longleftrightarrow$	*Rewriting*	$\longleftrightarrow$	*Proof*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*	$\longleftrightarrow$	*Logical Structure*

In particular, *concurrent computations* in our Petri net example

coincide with linear logic *proofs*.

Exercises

Ex.20.1. Show in detail that, given a Petri net N, the inference system given in pages 39–40 of Lecture 19 to generate all its concurrent computations is *equivalent* to the specialization to the theory $R(N)$ of the general inference system for rewriting logic.

Reachability Models

Given a general rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, a *reachability model* of $\mathcal{R} = (\Sigma, E, \phi, R)$ is a pair $(A, \rightarrow_A)$ with A a (Σ, E)-algebra and $\rightarrow_A = \{\rightarrow_{A,k}\}_{k \in K}$ a K-indexed family of binary relations, with $\rightarrow_{A,k} \subseteq A_k^2$ such that:

1. **Reflexivity and Transitivity**: for each $k \in K$ the relation $\rightarrow_{A,k}$ is reflexive and transitive;
2. **Congruence**: for each $f : k_1 \dots k_n \longrightarrow k$ in Σ such that $\{1, \dots, n\} - \phi(f) = \{i_1, \dots, i_m\} \neq \emptyset$, whenever we have $a_1 \in A_{k_1}, \dots, a_n \in A_{k_n}$ and for $1 \leq j \leq m$ we have $a_{i_j} \rightarrow_{A_{k_{i_j}}} a'_{i_j}$ then we also have,

$$f_A(a_1, \dots, a_{i_1}, \dots, a_{i_m}, \dots, a_n) \rightarrow_{A,k} f_A(a_1, \dots, a'_{i_1}, \dots, a'_{i_m}, \dots, a_n)$$

3. **Replacement**: for each rewrite rule in R,

$$l : (\forall X)\, t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l')$$

with, say t, t' of kind k, and w_l, w_l' of kind k_l, and for each assignment $a : X \longrightarrow A$ such that: (i) $\bigwedge_i \overline{a}(u_{i)} = \overline{a}(u_i')$, (ii) $\bigwedge_j \overline{a}(v_j) : s_j$, and (iii) $\bigwedge_l \overline{a}(w_l) \rightarrow_{A,k_l} \overline{a}(w_l')$, we have,

$$\overline{a}(t) \rightarrow_{A,k} \overline{a}(t')$$

The Initial Model $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$

The most obvious reachabilty model for a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ is the model $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$, where, by definition,

$$[t] \rightarrow_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad \mathcal{R} \vdash t \longrightarrow t'$$

This is indeed a reachability model, because all the requirements are guaranteed by $T_{\Sigma/E}$ being a (Σ, E)-algebra and by the inference rules of rewriting logic.

Given two reachability models $(A, \rightarrow_A)$ and $(B, \rightarrow_B)$ of $\mathcal{R} = (\Sigma, E, \phi, R)$ a $\mathcal{R}$-*homomorphism* $h : (A, \rightarrow_A) \longrightarrow (B, \rightarrow_B)$ is a (Σ, E)-homomorphism $h : A \longrightarrow B$ such that for each $k \in K$, $a \rightarrow_{A,k} a'$ implies $h_k(a) \rightarrow_{B,k} h_k(a')$.

The key point about $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ is that, as shown in Bruni and Meseguer "Generalized Rewrite Theories," in Proc. ICALP'03, Springer LNCS 2719, 252–266, 2003 for an equivalent formulation we have:

Theorem. For $\mathcal{R} = (\Sigma, E, \phi, R)$ a rewrite theory, $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ is an *initial* reachability model.

Therefore, when reasoning about a concurrent system specified by a rewrite theory $\mathcal{R}$, for example as a system module in Maude, we will view $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ as the *standard model* specified by $\mathcal{R}$, that is, as the mathematical model denoted by the specification $\mathcal{R}$. In other words, the initial algebra semantics of equational logic generalizes in a natural way to an initial reachability model semantics for rewriting logic.

Verification of Declarative Concurrent Programs

We are now ready to discuss the subject of *verification of declarative concurrent programs*, and, more specifically, the verification of properties of Maude *system modules*, that is, of declarative concurrent programs that are *rewrite theories*.

There are two levels of specification involved: (1) a *system specification* level, provided by the rewrite theory and yielding an *initial model* for our program; and (2) a *property specification* level, given by some property (or properties) φ that we want to prove about our program. To say that our program *satisfies* the property φ then means exactly to say that its initial model does.

Verification of Declarative Concurrent Programs (II)

Specifically, we have considered the *reachability* initial model, $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ of a rewrite theory $\mathcal{R}$.

The question then becomes, which *language* shall we use to express the *properties* φ that we want to prove hold in the model $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$? That is, how should we express relevant properties φ such that,

$$(T_{\Sigma/E}, \rightarrow_{\mathcal{R}}) \models \varphi.$$

One possibility would to use a *first-order language* defined by the signature Σ together with a family of binary transition relations, one for each kind k in Σ.

Verification of Declarative Concurrent Programs (IV)

In particular (see 2001 Lecture Notes of CS 376 at UIUC) given a rewrite theory $\mathcal{R}$, the *modal logic* $\mathcal{M}(\mathcal{R})$, expressing properties based on *necessity*, $\Box\varphi$, and *possibility*, $\Diamond\varphi$, can be regarded as a *sublanguage* of such a first-order language.

But not all properties of interest are expressible in $\mathcal{M}(\mathcal{R})$. For example, properties involving *fairness*, and other properties related to the *infinite behavior* of a system typically are not expressible in $\mathcal{M}(\mathcal{R})$.

For such properties we can use some kind of *temporal logic*. We will give particular attention to *linear temporal logic* (LTL) because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

The Syntax of $LTL(AP)$

Given a set AP of *atomic propositions*, we define the formulae of the *propositional linear temporal logic* $LTL(AP)$ inductively as follows:

- **True**: $\top \in LTL(AP)$.
- **Atomic propositions**: If $p \in AP$, then $p \in LTL(AP)$.
- **Next operator**: If $\varphi \in LTL(AP)$, then $\bigcirc\varphi \in LTL(AP)$.
- **Until operator**: If $\varphi, \psi \in LTL(AP)$, then $\varphi \,\mathcal{U}\, \psi \in LTL(AP)$.
- **Boolean connectives**: If $\varphi, \psi \in LTL(AP)$, then the formulae $\neg\varphi$, and $\varphi \vee \psi$ are in $LTL(AP)$.

The Syntax of $LTL(AP)$ (II)

Other LTL connectives can be defined in terms of the above minimal set of connectives as follows:

- Other Boolean connectives:
  - **False**: $\bot = \neg\top$
  - **Conjunction**: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - **Implication**: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.

- Other temporal operators:
  - **Eventually**: $\Diamond\varphi = \top\,\mathcal{U}\,\varphi$
  - **Henceforth**: $\Box\varphi = \neg\Diamond\neg\varphi$
  - **Release**: $\varphi\,\mathcal{R}\,\psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$
  - **Unless**: $\varphi\,\mathcal{W}\,\psi = (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$
  - **Leads-to**: $\varphi \rightsquigarrow \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - **Strong implication**: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - **Strong equivalence**: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.

Kripke Structures

Kripke structures are the natural models for propositional temporal logic. Essentially, a Kripke structure is a (total) *unlabeled transition system* to which we have added a collection of unary state predicates on its set of states.

A binary relation $R \subseteq A \times A$ on a set A is called *total* iff for each $a \in A$ there is at least one $a' \in A$ such that $(a, a') \in R$. If R is not total, it can be made total by defining
$R^\bullet = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A \; (a, a') \in R\}$.

Kripke Structures (II)

A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set, called the set of *states*, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A, called the *transition relation*, and $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the *labeling function*, associating to each state $a \in A$ the set $L(a)$ of those *atomic propositions* in AP that *hold* in the state a.

How can we associate a Kripke structure to a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$? We just need to make explicit two things: (1) the intended *kind* k of states in the signature Σ; and (2) the relevant *state predicates*, that is, the relevant set AP of atomic propositions.

Kripke Structures (III)

When we fix a kind k as the kind of states, our associated Kripke structure is obtained from the initial reachability model $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ by defining the set of states as $T_{\Sigma/E,k}$ and the (total!) transition relation as $(\rightarrow^1_{\mathcal{R}})^\bullet$, the totalization of the *one-step* rewrite relation $\rightarrow^1_{\mathcal{R}}$ on $T_{\Sigma/E,k}$. By definition, $[t] \rightarrow^1_{\mathcal{R}} [t']$ iff there is a proof of $\mathcal{R} \vdash t \longrightarrow t'$ using exactly one application of the **Replacement** inference rule where, furthermore, all the rewrites in the substitution part are identity rewrites obtained by **Reflexivity**.

We will explain later in this lecture how the remaining part of the Kripke structure, namely the labeling function specifying the state predicates, can also be defined for a rewrite theory $\mathcal{R}$ for the desired state predicates.

The Semantics of $LTL(AP)$

The semantics of the temporal logic LTL is defined by means of a *satisfaction relation*

$$\mathcal{A}, a \models \varphi$$

between a Kripke structure $\mathcal{A}$ having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in LTL(AP)$. Specifically, $\mathcal{A}, a \models \varphi$ holds iff for each path $\pi \in Path(\mathcal{A})_a$ the *path satisfaction relation*

$$\mathcal{A}, \pi \models \varphi$$

holds, where we define the set $Path(\mathcal{A})_a$ of *computation paths* starting at state a as the set of functions of the form $\pi : \mathbb{N} \longrightarrow A$ such that $\pi(0) = a$ and, for each $n \in \mathbb{N}$, we have $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

The Semantics of $LTL(AP)$ (II)

We can define the path satisfaction relation (for any path, beginning at any state) inductively as follows:

- We always have $\mathcal{A}, \pi \models_{LTL} \top$.
- For $p \in AP$,

$$\mathcal{A}, \pi \models_{LTL} p \quad \Leftrightarrow \quad p \in L(\pi(0)).$$

- For $\bigcirc\varphi \in LTL(A)$,

$$\mathcal{A}, \pi \models_{LTL} \bigcirc\varphi \quad \Leftrightarrow \quad \mathcal{A}, s; \pi \models_{LTL} \varphi,$$

  where $s : \mathbb{N} \longrightarrow \mathbb{N}$ is the successor function.

- For $\varphi \,\mathcal{U}\, \psi \in LTL(\mathcal{A})$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \,\mathcal{U}\, \psi \quad \Leftrightarrow$$

$(\exists n \in \mathbb{N})\ ((\mathcal{A}, s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N})\ m < n\ \Rightarrow \mathcal{A}, s^m; \pi \models_{LTL} \varphi))$.

- For $\neg\varphi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \neg\varphi \quad \Leftrightarrow \quad \mathcal{A}, \pi \not\models_{LTL} \varphi.$$

- For $\varphi \vee \psi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \quad \Leftrightarrow$$

$$\mathcal{A}, \pi \models_{LTL} \varphi \quad \text{or} \quad \mathcal{A}, \pi \models_{LTL} \psi.$$

The LTL Module

The LTL syntax, in a typewriter approximation of the mathematical syntax, is supported in Maude by the following `LTL` functional module (in the file `model-checker.maude`).

```
fmod LTL is
  sorts Prop Formula .
  subsort Prop < Formula .

*** primitive LTL operators
  ops True False : -> Formula [ctor format (g o)] .
  op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _/\_ : Formula Formula -> Formula [comm ctor gather (E e)
                                        prec 55 format (d r o d)] .
  op _\/_ : Formula Formula -> Formula [comm ctor gather (E e)
                                        prec 59 format (d r o d)] .
```

```
  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _U_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)]
  op _R_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)]

*** defined LTL operators
  op _->_ : Formula Formula -> Formula [gather (e E)
                                        prec 65 format (d r o d)] .
  op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
  op <>_ : Formula -> Formula [prec 53 format (r o d)] .
  op []_ : Formula -> Formula [prec 53 format (r d o d)] .
  op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
  op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
  op _=>_ : Formula Formula -> Formula [gather (e E) prec 65
                                        format (d r o d)] .
  op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .

  vars f g : Formula .
```

```
  eq f -> g = ~ f \/ g .
  eq f <-> g = (f -> g) /\ (g -> f) .
  eq <> f = True U f .
  eq [] f = False R f .
  eq f W g = (f U g) \/ [] f .
  eq f |-> g = [](f -> (<> g)) .
  eq f => g = [] (f -> g) .
  eq f <=> g = [] (f <-> g) .

*** negative normal form
  eq ~ True = False .
  eq ~ False = True .
  eq ~ ~ f = f .
  eq ~ (f \/ g) = ~ f /\ ~ g .
  eq ~ (f /\ g) = ~ f \/ ~ g .
  eq ~ O f = O ~ f .
  eq ~(f U g) = (~ f) R (~ g) .
  eq ~(f R g) = (~ f) U (~ g) .
```

```
endfm
```

The LTL Module (II)

Note the subsort `Prop` of `Formula`, corresponding to the set AP of atomic propositions. For the moment this is left unspecified. We will explain in what follows how such atomic propositions are defined for a given system module `M`.

Note that the nonconstructor connectives have been defined in terms of more basic constructor connectives in the first set of equations. But since there are good reasons to put an LTL formula in *negative normal form* by pushing the negations next to the atomic propositions (this is specified by the second set of equations) we need to consider also the *duals* of the basic connectives $\top$, $\bigcirc$, $\mathcal{U}$, and $\vee$ as constructors. That is, we need to also have as constructors the dual connectives: $\bot$, $\mathcal{R}$, and $\wedge$ (note that $\bigcirc$ is self-dual).

Associating Kripke structures to Rewrite Theories

Since the models of temporal logic are Kripke structures, we need to explain how we can associate a Kripke structure to the rewrite theory specified by a Maude system module `M`.

Indeed, we associate a Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by a Maude system module `M` by making explicit two things: (1) the intended *kind* k of states in the signature Σ; and (2) the relevant *state predicates*, that is, the relevant set AP of atomic propositions.

In general, the state predicates need not be part of the *system specification* and therefore they need not be specified in our system module `M`. They are typically part of the *property specification*.

Associating Kripke structures to Rewrite Theories (II)

This is because the state predicates need not be related to the operational semantics of `M`: they are just certain *predicates* about the states of the system specified by `M` that are needed to specify some *properties*.

Therefore, after choosing a given kind, say `[Foo]`, in `M` as our kind for states we can specify the relevant state predicates in a module `M-PREDS` protecting `M` according to the following general pattern:

```
mod M-PREDS is protecting M .
  including SATISFACTION .
  subsort Foo < State .
  ...
endm
```

Associating Kripke structures to Rewrite Theories (III)

Where the dots '. . .' indicate the part in which the syntax and semantics of the relevant state predicates is specified, as further explained in what follows. The module `SATISFACTION` (which is contained in the file `model-checker.maude`) is very simple, and has the following specification:

```
fmod SATISFACTION is
  protecting LTL .
  sort State .
  op _|=_ : State Formula ~> Bool .
endfm
```

where the sort `State` is unspecified. However, by importing `SATISFACTION` into `M-PREDS` and giving the subsort declaration

Associating Kripke structures to Rewrite Theories (IV)

```
subsort Foo < State .
```

all terms of sort `Foo` in `M` are also made terms of sort `State`. Note that we then have the kind identity, `[Foo]=[State]`.

The operator

```
op _|=_ : State Formula ~> Bool .
```

is crucial to define the semantics of the relevant state predicates in `M-PREDS`. Each such state predicate is declared as an operator of sort `Prop`.

In standard LTL propositional logic the set AP of atomic propositions is assumed to be a set of *constants*.

Associating Kripke structures to Rewrite Theories (V)

In Maude we can define *parametric* state predicates, that is, operators of sort `Prop` which need not be constants, but may have one or more sorts as parameter arguments. We then define the *semantics* of such state predicates (when the predicate holds) by appropriate equations.

We can illustrate all this by means of a simple mutual exclusion example. Suppose that our original system module `M` is the following module `MUTEX`, in which two processes, one named `a` and another named `b`, can be either waiting or in their critical section, and take turns accessing their critical section by passing each other a different *token* (either `$` or `*`).

Associating Kripke structures to Rewrite Theories (VI)

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf .
  op __ : Conf Conf -> Conf [assoc comm id: none] .
  ops a b : -> Name .
  ops wait critical : -> Mode .
  op [_,_] : Name Mode -> Proc .
  ops * $ : -> Token .
  rl [a-enter] : $ [a,wait] => [a,critical] .
  rl [b-enter] : * [b,wait] => [b,critical] .
  rl [a-exit] : [a,critical] => [a,wait] * .
  rl [b-exit] : [b,critical] => [b,wait] $ .
endm
```

Associating Kripke structures to Rewrite Theories (VII)

Our obvious kind for states is the kind `[Conf]` of configurations. In order to state the desired safety and liveness properties we need state predicates telling us whether a process is waiting or is in its critical section. We can make these predicates *parametric* on the name of the process and define their semantics as follows:

```
mod MUTEX-PREDS is protecting MUTEX .  including SATISFACTION .
  subsort Conf < State .
  op crit : Name -> Prop .
  op wait : Name -> Prop .
  var N : Name .
  var C : Conf .
  eq [N,critical] C |= crit(N) = true .
  eq [N,wait] C |= wait(N) = true .
endm
```

Associating Kripke structures to Rewrite Theories (VIII)

Note the two equations, defining when each of the two parametric state predicates holds in a given state.

The above example illustrates a *general method* by which desired state predicates for a module `M` are defined in a *protecting* extension, say `M-PREDS`, of `M` which imports `SATISFACTION`.

One specifies the desired states by choosing a sort in `M` and declaring it as a subsort of `State`. One then defines the syntax of the desired state predicates as operators of sort `Prop`, and defines their semantics by means of a set of equations that specify for what states a given state predicate evaluates to `true`.

Associating Kripke structures to Rewrite Theories (IX)

We assume that those equations, when added to those of `M`, are (ground) Church-Rosser and terminating.

Note that *only the cases when a predicate holds* need to be specified: given a state t and a, possibly parametric, state predicate $p(u_1, \ldots, u_n)$, when the ground expression $t \models p(u_1, \ldots, u_n)$ cannot be simplified to $true$, then the predicate does *not* hold. This means that to specify the semantics of the state predicates it is enough to give (possibly conditional) equations of the general form,

$$t \models p(v_1, \ldots, v_n) = true \;\; if \;\; C.$$

There is in principle no need to specify when a state predicate is *false*.

Associating Kripke structures to Rewrite Theories (X)

However, if so desired one can specify both the true and false cases. This can always be easily done either by using the `[owise]` attribute, or by giving (possibly conditional) equations of the more general form,

$$t \models p(v_1, \ldots, v_n) = bexp \ \ if \ C,$$

where *bexp* is an arbitrary Boolean expression.

We are now ready to associate to a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind k of states and with state predicates Π defined by means of equations D in a protecting extension `M-PREDS`) a Kripke structure whose atomic predicates are specified by the set

Associating Kripke structures to Rewrite Theories (XI)

$$AP_\Pi = \{\theta(p) \mid p \in \Pi,\ \theta \text{ ground substitution}\},$$

where, by convention, we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \ldots, x_n))$.

This defines a labeling function L_Π on the set of states $T_{\Sigma/E,k}$ assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions,

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid (E \cup D) \vdash\ (\forall\, \emptyset)\ t \models \theta(p) = true\}.$$

The Kripke structure we are interested in is then

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$$

Associating Kripke structures to Rewrite Theories (XII)

Where $(\rightarrow^1_{\mathcal{R}})^\bullet$ denotes the total relation extending the one-step $\mathcal{R}$-rewriting relation $\rightarrow^1_{\mathcal{R}}$ among states of kind k, that is, $[t] \rightarrow^1_{\mathcal{R}} [t']$ holds iff there are $u \in [t]$ and $u' \in [t']$ such that u' is the result of applying one of the rules in R to u at some position.

Under the usual assumptions that E is (ground) Church-Rosser and terminating (perhaps modulo some axioms A contained in E) and R is (ground) coherent relative to E, u can always be chosen to be the canonical form of t under the equations E.

Decidability of Propositional LTL

It is well-known that, for any *computable* Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, any state $a \in A$ such that the set

$$Reach_{\mathcal{A}}(a) = \{x \in A \mid \exists \pi \in Path(\mathcal{A})\ \exists n \in \mathbb{N}\ \ s.t.\ \pi(0) = a \wedge \pi(n) = x\}$$

of states *reachable* from a in $\mathcal{A}$ is *finite*, and any LTL formula $\varphi \in LTL(AP)$, where $L : A \longrightarrow \mathcal{P}(AP)$, there is a *decision procedure* that can *effectively decide* the satisfaction relation,

$$\mathcal{A}, a \models_{LTL} \varphi.$$

Furthermore, if $\mathcal{A}, a \not\models_{LTL} \varphi$, the decision procedure will exhibit a *counterexample*, that is, a path not satisfying φ.

Decidability of Propositional LTL (II)

A decision procedure of this kind is called a *model checking algorithm*, since it checks whether φ holds in the model $\mathcal{A}$ with initial state a. Detailed discussion of such algorithms for a variety of temporal logics such as LTL, CTL, and CTL^* is beyond the scope of this course; see the excellent text "Model Checking" by Clark, Grumberg, and Peled. There are two rough classes of model checking algorithms:

- *explicit-state* model checking algorithms, that explicitly search the state space of $\mathcal{A}$ to find a counterexample;
- *symbolic model checking* algoritms, that use a symbolic Boolean representation of *sets of states* (typically BDDs) to compute the fixpoint of the transition relation, i.e., the set $Reach_{\mathcal{A}}(a)$.

The Maude Model Checker

Suppose that, given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, we have:

- chosen a kind k in `M` as our kind of states;
- defined some state predicates Π and their semantics in a module, say `M-PREDS`, protecting `M` by the method already explained in this lecture.

Then, as explained earlier, this defines a Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$ on the set of atomic propositions AP_Π. Given an initial state $[t] \in T_{\Sigma/E,k}$ and an LTL formula $\varphi \in LTL(AP_\Pi)$ we would like to have a procedure to decide the satisfaction relation,

The Maude Model Checker (II)

$$\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \models \varphi.$$

By applying the general LTL decidability results to our Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi}$, this satisfaction relation becomes decidable if two conditions hold:

1. The set of states in $T_{\Sigma/E,k}$ that are *reachable* from $[t]$ by rewriting is *finite*.

2. The rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by `M` plus the equations D defining the predicates Π are such that:

The Maude Model Checker (III)

- both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A, and
- R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under these assumptions, both the state predicates Π and the transition relation $\rightarrow^1_{\mathcal{R}}$ are *computable* and, given the finite reachability assumption, we can then settle the above satisfaction problem using a *model checking procedure*. Specifically, Maude uses an on-the-fly LTL model checking procedure of the style described by Clark, Grumberg, and Peled.

The Maude Model Checker (III)

The basis of this procedure is the following. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω-language is exactly that of the behaviors satisfying φ. We can then reduce the satisfaction problem

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$$

to the *emptiness problem* of the language accepted by the *synchronous product* of $B_{\neg\varphi}$ and (the Büchi automaton associated to) $(\mathcal{K}(\mathcal{R}, k)_\Pi, [t])$. The formula φ is satisfied iff such a language is empty. The model checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by the synchronous product.

The Maude Model Checker (IV)

This makes clear our interest in obtaining the *negative normal form* of a formula $\neg\varphi$, since we need it to build the Büchi automaton $B_{\neg\varphi}$.

For efficiency purposes we need to make $B_{\neg\varphi}$ as small as possible. The following module `LTL-SIMPLIFIER` (also in the `model-checker.maude` file) tries to further simplify the negative normal form of the formula $\neg\varphi$ in the hope of generating a smaller Büchi automaton $B_{\neg\varphi}$. This module is optional (the user may choose to include it or not when doing model checking) but tends to help building a smaller $B_{\neg\varphi}$.

The Maude Model Checker (V)

```
fmod LTL-SIMPLIFIER is
  including LTL .

*** The simplifier is based on:
***   Kousha Etessami and Gerard J. Holzman,
***   "Optimizing Buchi Automata", p153-167, CONCUR 2000, LNCS 1877.
*** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
          PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
```

```
op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op O_ : PE-Formula -> PE-Formula [ctor ditto] .
op O_ : PU-Formula -> PU-Formula [ctor ditto] .
op O_ : PureFormula -> PureFormula [ctor ditto] .
op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .
```

```
  vars p q r s : Formula .
  var pe : PE-Formula .
  var pu : PU-Formula .
  var pr : PureFormula .

*** Rules 1, 2 and 3; each with its dual.
  eq (p U r) /\ (q U r) = (p /\ q) U r .
  eq (p R r) \/ (q R r) = (p \/ q) R r .
  eq (p U q) \/ (p U r) = p U (q \/ r) .
  eq (p R q) /\ (p R r) = p R (q /\ r) .
  eq True U (p U q) = True U q .
  eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
  eq p U pe = pe .
  eq p R pu = pu .
```

```
*** An extra rule in the same style.
  eq O pr = pr .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
***   "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.

*** Four pairs of duals.
  eq O p /\ O q = O (p /\ q) .
  eq O p \/ O q = O (p \/ q) .
  eq O p U O q = O (p U q) .
  eq O p R O q = O (p R q) .
  eq True U O p = O (True U p) .
  eq False R O p = O (False R p) .
  eq (False R (True U p)) \/ (False R (True U q)) = False R (True U (p
  eq (True U (False R p)) /\ (True U (False R q)) = True U (False R (p
```

```
*** <= relation on formula
  op _<=_ : Formula Formula -> Bool [prec 75] .

  eq p <= p = true .
  eq False <= p  = true .
  eq p <= True = true .

  ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
  ceq p <= (q \/ r) = true if p <= q .
  ceq (p /\ q) <= r = true if p <= r .
  ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

  ceq p <= (q U r) = true if p <= r .
  ceq (p R q) <= r = true if q <= r .
  ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
  ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
  ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
```

```
  ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

*** conditional rules depending on <= relation
  ceq p /\ q = p if p <= q .
  ceq p \/ q = q if p <= q .
  ceq p /\ q = False if p <= ~ q .
  ceq p \/ q = True if ~ p <= q .
  ceq p U q = q if p <= q .
  ceq p R q = q if q <= p .
  ceq p U q = True U q if p =/= True /\ ~ q <= p .
  ceq p R q = False R q if p =/= False /\ q <= ~ p .
  ceq p U (q U r) = q U r if p <= q .
  ceq p R (q R r) = q R r if q <= p .
endfm
```

The Maude Model Checker (VI)

Suppose that all the requirements listed above to perform model checking are satisfied. How do we then model check a given LTL formula in Maude for a given initial state $[t]$ in a module `M`? We define a new module, say `M-CHECK`, according to the following pattern:

```
mod M-CHECK is
  protecting M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER . *** optional
  op init : -> k .           *** optional
  eq init = t .              *** optional
endm
```

The declaration of a constant `init` of the kind of states is not

necessary: it is a matter of convenience, since the initial state `t` may be a large term.

The Maude Model Checker (VII)

The module MODEL-CHECKER is as follows.

```
fmod MODEL-CHECKER is protecting QID .
including SATISFACTION .

*** transitions and results
  sorts RuleName Transition TransitionList ModelCheckResult .
  subsort Qid < RuleName .
  subsort Transition < TransitionList .
  subsort Bool < ModelCheckResult .
  ops unlabeled deadlock : -> RuleName .
  op {_,_} : State RuleName -> Transition .
  op nil : -> TransitionList [ctor] .
  op __ : TransitionList TransitionList -> TransitionList [ctor assoc
  op counterexample : TransitionList TransitionList -> ModelCheckResul
```

```
  op modelCheck : State Formula ~> ModelCheckResult [special ( ... )]
endfm
```

The Maude Model Checker (VIII)

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes a state and an LTL formula and returns either the Boolean `true` if the formula is satisfied, or a counterexample when it is not satisfied.

Let us illustrate the use of this operator with our `MUTEX` example. Following the pattern described above, we can define the module

```
mod MUTEX-CHECK is
  including MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a,wait] [b,wait] .
  eq initial2 = * [a,wait] [b,wait] .
```

```
endm
```

The Maude Model Checker (X)

We are then ready to model check different LTL properties of `MUTEX`. The first obvious property to check is mutual exclusion:

```
Maude> red modelCheck(initial1,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []~ (crit(a) /\ crit(b)))
rewrites: 18 in 10ms cpu (10ms real) (1800 rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []~ (crit(a) /\ crit(b)))
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XII)

We can also model check the strong liveness property that if a process waits infinitely often, then it is in its critical section infinitely often:

```
Maude> red modelCheck(initial1,([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(a) -> []<> crit
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial1,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(b) -> []<> crit
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(a)) -> ([] <> crit(a))) .
```

```
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(a) -> []<> crit
rewrites: 68 in 10ms cpu (10ms real) (6800 rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(b) -> []<> crit
rewrites: 68 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XIII)

Of course, not all properties are true. Therefore, instead of a success we can get a *counterexample* showing why a property fails. Suppose that we want to check whether, beginning in the state `initial1`, process `b` will always be waiting. We then get the counterexample:

```
Maude> red modelCheck(initial1,[] wait(b)) .
reduce in MUTEX-CHECK : modelCheck(initial1, []wait(b)) .
rewrites: 14 in 10ms cpu (10ms real) (1400 rewrites/second)
result ModelCheckResult:
  counterexample({$ [a,wait] [b,wait],'a-enter}
                 {[a,critical] [b,wait],'a-exit}
                 {* [a,wait] [b,wait],'b-enter},
                 {[a,wait] [b,critical],'b-exit}
                 {$ [a,wait] [b,wait],'a-enter}
```

```
{[a,critical] [b,wait],'a-exit}
{* [a,wait] [b,wait],'b-enter})
```

The Maude Model Checker (XIV)

The main counterexample term constructors are:

```
op {_,_} : State RuleName -> Transition .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList [ctor assoc
op counterexample : TransitionList TransitionList -> ModelCheckResul
```

A counterexample is a pair consisting of two lists of transitions: the first is a finite path beginning in the initial state, and the second describes a loop. This is because, if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a path of transitions followed by a cycle. Note that each transition is represented as a *pair*, consisting of a state and the label of the rule applied to reach the next state.

Model Checking TOK-RING

Consider the following TOK-RING module,

```
(fth NZNAT* is
   protecting NAT .
   op * : -> NzNat .
 endfth)

(fmod NAT/(N :: NZNAT*) is
   sort Nat/(N) .
   op `[_`] : Nat -> Nat/(N) .
   op _+_ : Nat/(N) Nat/(N) -> Nat/(N) .
   op _*_ : Nat/(N) Nat/(N) -> Nat/(N) .
   vars I J : Nat .
   ceq [I] = [I rem *] if I >= * .
   eq [I] + [J] = [I + J] .
```

```
   eq [I] * [J] = [I * J] .
 endfm)

(omod TOK-RING(N :: NZNAT*) is
   protecting NAT/(N) .
   sort Mode .
   subsort Nat/(N) < Oid .
   ops wait critical : -> Mode .
   msg tok : Nat/(N) -> Msg .
   op init : -> Configuration .
   op make-init : Nat/(N) -> Configuration .
   class Proc | mode : Mode .
   var I : Nat .
   ceq init = tok([0]) make-init([I]) if s(I) := * .
   ceq make-init([s(I)])
     = < [s(I)] : Proc | mode : wait > make-init([I])
     if I < * .
   eq make-init([0]) = < [0] : Proc | mode : wait > .
```

```
  rl [enter] : tok([I]) < [I] : Proc | mode : wait >
    => < [I] : Proc | mode : critical > .
  rl [exit] : < [I] : Proc | mode : critical >
    => < [I] : Proc | mode : wait > tok([s(I)]) .
endom)
```

Model Checking TOK-RING (II)

The TOK-RING module satisfies the following two properties:

- *mutual exclusion*, and
- *guaranteed reentrance*, that is:
  - each process eventually reaches its critical section, and
  - it does so again after $2 \times n$ steps.

There isn't a single LTL formula stating each of these properties: they are *parametric* on n. However, in Full Maude we can specify these properties by parametic formula definitions as follows:

Model Checking TOK-RING (III)

```
(omod CHECK-TOK-RING(N :: NZNAT*) is
  inc TOK-RING(N) .
  inc MODEL-CHECKER .
  subsort Configuration < State .

  op inCrit : Nat/(N) -> Prop .
  op twoInCrit : -> Prop .

  var  I : Nat .
  vars X Y : Nat/(N) .
  var  C : Configuration .
  var  F : Formula .

  eq < X : Proc | mode : critical > C |= inCrit(X) = true .
  eq < X : Proc | mode : critical > < Y : Proc | mode : critical > C
```

```
     |= twoInCrit = true .


op guaranteedReentrance : -> Formula .
op allProcessesReenter : Nat -> Formula .
op nextIter_ : Formula -> Formula .
op nextIterAux : Nat Formula -> Formula .

ceq guaranteedReentrance = allProcessesReenter(I) if s(I) := * .

eq allProcessesReenter(s(I))
  = (<> inCrit([s(I)])) /\
     [] (inCrit([s(I)]) -> (nextIter inCrit([s(I)]))) /\
    allProcessesReenter(I) .
eq allProcessesReenter(0) = (<> inCrit([0])) /\
     [] (inCrit([0]) -> (nextIter inCrit([0]))) .

eq nextIter F = nextIterAux(2 * *, F) .
```

```
 eq nextIterAux(s I, F) = O nextIterAux(I, F) .
 eq nextIterAux(O, F) = F .

endom)
```

Model Checking TOK-RING (IV)

We cannot model check these properties directly in their *parameterized* form. However, for each nozero value n we can check the corresponding *instance* of these properties. For example, for $n = 5$ we define in Full Maude the *view*,

```
(view 5 from NZNAT* to NAT is
   op * to term 5 .
 endv)
```

Then we can model check the mutual exclusion property for 5 processes as follows:

```
(red in CHECK-TOK-RING(5) : modelCheck(init,[] ~ twoInCrit) .)
result Bool :
  true
```

Model Checking TOK-RING (V)

In the same way, we can model check the *guaranteed reentrance* property for $n = 5$ by giving to Full Maude the command,

```
(red in CHECK-TOK-RING(5) : modelCheck(init,[] guaranteedReentrance) .)
result Bool :
  true
```

Verification of Concurrent Imperative Programs

In the case of *deterministic* programs, we first studied the verification of *declarative* deterministic programs such as Maude functional modules. Then, in a sense, we *reduced* to this case the verification of *imperative* programs.

Indeed, we can specify the *semantics* of a deterministic imperative language $\mathcal{L}$ as an *equational theory* $\mathcal{E}(\mathcal{L})$ (in fact, a Maude functional module).

Then, reasoning about the correctness of imperative programs in $\mathcal{L}$ reduces (perhaps through decomposition by means of a Hoare logic) to *proving inductive properties* satisfied by the initial model $T_{\mathcal{E}(\mathcal{L})}$.

Verification of Concurrent Imperative Programs (II)

What should the analogous situation be in the case of *concurrent* imperative programs? We should of course specify the *semantics* of a concurrent imperative language $\mathcal{L}$ as a *rewrite theory* $\mathcal{R}(\mathcal{L})$ (in fact, a Maude system module).

Then, the correctness of imperative programs in $\mathcal{L}$ can be reduced to *proving inductive properties* satisfied by the initial model $(T_{\Sigma_{\mathcal{L}}/E_{\mathcal{L}}}, \rightarrow_{\mathcal{R}_{\mathcal{L}}})$. If such properties are specified in *temporal logic*, then we can use methods such as model checking or deductive proof.

We can illustrate this general method by definig the rewriting logic semantics of a simple parallel language called `PARALLEL`.

The Rewriting Semantics of PARALLEL

```
*** A simple parallel language and its rewriting logic semantics.
*** Extends an even simpler language presented in ‘‘The Maude LTL
*** Model Checker’’ by Eker, Meseguer, and Sridaranarayanan,
*** in Proc. WRLA’02, ENTCS Vol. 71, Elsevier, 2002.

fmod MEMORY is inc INT .  inc QID .
  sorts Memory Bool? Int? .
  subsorts Bool < Bool? . subsorts Int < Int? .
  op null : -> Int? .
  op none : -> Memory .
  op __ : Memory Memory -> Memory [assoc comm id: none] .
  op [_,_] : Qid Int? -> Memory .
  op _in_ : Qid Memory -> Bool? .
  var Q : Qid .   var M : Memory .  var N? : Int? .
  eq null + N? = null .
```

```
  eq null * N? = null .
  eq Q in [Q,N?] M = true .
endfm


***(Equality test comparing the contents of a named memory location to
an Int? value.)

fmod TESTS is
  inc MEMORY .
  sort Test .
  op _=_ : Qid Int? -> Test .
  op eval : Test Memory -> Bool .
  var Q : Qid .
  var M : Memory .
  vars N? N'? : Int? .
  eq eval(Q = N?, [Q, N'?] M) = N? == N'? .
  ceq eval(Q = N?, M) = N? == null if Q in M =/= true .
```

```
endfm

***(Syntax for arithmetic expressions, and their evaluation semantics.
avoid evaluation of expressions by themselves, which would happen even w
a memory for integer subexpressions if we keep the usual syntax, the ope
+ and * are specified as constructors with syntax +' and *' )

fmod EXPRESSION is
  inc MEMORY .
  sort Expression .
  subsorts Qid Int? < Expression .
  op _+'_ : Expression Expression -> Expression [ctor] .
  op _*'_ : Expression Expression -> Expression [ctor] .
  op eval : Expression Memory -> Int? .

  var Q : Qid .
  var M : Memory .
  vars N N' : Int .
```

```
  var N? : Int? .
  vars E E' : Expression .

  eq eval(N?, M) = N? .
  eq eval(Q, [Q, N?] M) = N? .
  ceq eval(Q,M) = null if Q in M =/= true .
  eq eval(E +' E', M) = eval(E,M) + eval(E',M) .
  eq eval(E *' E', M) = eval(E,M) * eval(E',M) .
endfm

***(Syntax for a trival sequential language.  We allow abstracting
out program fragments as elements of sorts LoopingUserStatement and
UserStatement.  Elements of sort LoopingUserStatement abstract out
potentially nonterminating program fragments, whereas elements of sort
UserStatement but not of sort LoopingUserStatement abstract out terminat
program fragments.)

fmod SEQUENTIAL is
```

```
  inc TESTS .
  inc EXPRESSION .

  sorts UserStatement LoopingUserStatement Program .
  subsort LoopingUserStatement < UserStatement < Program .
  op skip : -> Program .
  op _;_ : Program Program -> Program [prec 61 assoc id: skip] .
  op _:=_ : Qid Expression -> Program .
  op if_then_fi : Test Program -> Program .
  op while_do_od : Test Program -> Program .
  op repeat_forever : Program -> Program .
endfm
```

The Rewriting Semantics of PARALLEL (II)

Using the above functional modules, we can then define our simple parallel language in a system module PARALLEL. The *global state* is a *triple* consisting of:

1. a "soup" (set) of processes;
2. the shared memory; and
3. a process identifier recording the last process that touched the memory or, in any event, performed some computation.

Processes themselves are *pairs* having a process identifier and a program.

The Rewriting Semantics of PARALLEL (III)

```
mod PARALLEL is
  inc SEQUENTIAL .
  inc TESTS .

  sorts Pid Process Soup MachineState .
  subsort Process < Soup .
  subsort Int < Pid .
  op [_,_] : Pid Program -> Process .
  op empty : -> Soup .
```

```
op _|_ : Soup Soup -> Soup [prec 61 assoc comm id: empty] .
op {_,_,_} : Soup Memory Pid -> MachineState .

vars P R : Program .
var S : Soup .
var U : UserStatement .
var L : LoopingUserStatement .
vars I J : Pid .
var M : Memory .
var Q : Qid .
vars N? X? : Int? .
var T : Test .
var E : Expression .

rl {[I, U ; R] | S, M, J} => {[I, R] | S, M, I} .

rl {[I, L ; R] | S, M, J} => {[I, L ; R] | S, M, I} .
```

```
  rl {[I, (Q := E) ; R] | S, [Q, X?] M, J} =>
       {[I, R] | S, [Q,eval(E,[Q, X?] M)] M, I} .

  crl {[I, (Q := E) ; R] | S, M, J} =>
       {[I, R] | S, [Q,eval(E,M)] M, I} if Q in M =/= true .

  rl {[I, if T then P fi ; R] | S, M, J} =>
       {[I, if eval(T, M) then P else skip fi ; R] | S, M, I} .

  rl {[I, while T do P od ; R] | S, M, J} =>
     {[I, if eval(T, M) then (P ; while T do P od) else skip fi ; R]
         | S, M, I} .

  rl {[I, repeat P forever ; R] | S, M, J} =>
       {[I, P ; repeat P forever ; R] | S, M, I} .
endm
```

Dekker's Mutex Algorithm

One of the earliest correct solutions to the mutual exclusion problem was given by Dekker with his algorithm. The algorithm assumes processes that execute concurrently on a shared memory machine and communicate with each other through shared variables.

There are two processes, `p1` and `p2`. Process 1 sets a Boolean variable `c1` to 1 to indicate that it wishes to enter its critical section. Process `p2` does the same with variable `c2`. If one process, after setting its variable to 1 finds that the variable of its competitor is 0, then it enters its critical section rightaway. In case of a tie (both variables set to 1) the tie is broken using a variable `turn` that takes values in $\{1, 2\}$.

Dekker's Mutex Algorithm (II)

The code of process 1 in `PARALLEL` is as follows,

```
repeat
  c1 := 1 ;
  while c2 = 1 do
    if turn = 2 then
      c1 := 0 ;
      while turn = 2 do skip od ;
      c1 := 1

    fi
  od ;
  crit ;
  turn := 2 ;
  c1 := 0 ;
```

```
  rem1
forever .
```

Dekker's Mutex Algorithm (III)

The code of process 2 is entirely symmetric:

```
repeat
  c2 := 1 ;
  while c1 = 1 do
    if turn = 1 then
      c2 := 0 ;
      while turn = 1 do skip od ;
      c2 := 1
    fi
  od ;
  crit ;
  turn := 1 ;
  c2 := 0 ;
  rem2
forever .
```

Dekker's Mutex Algorithm (IV)

We can then define the two processes for Dekker's algorithm and the desired initial state in the following module extending `PARALLEL`. Note that we assume that `crit` does terminate, whereas `rem` may not.

```
mod DEKKER is
  inc PARALLEL .
  subsort Int < Pid .
  op crit : -> UserStatement .
  op rem : -> LoopingUserStatement .
  ops p1 p2 : -> Program .
  op initialMem : -> Memory .
  op initial : -> MachineState .
```

```
eq p1 =
      repeat
        'c1 := 1 ;
        while 'c2 = 1 do
          if 'turn = 2 then
            'c1 := 0 ;
            while 'turn = 2 do skip od ;
            'c1 := 1
          fi
        od ;
        crit ;
        'turn := 2 ;
        'c1 := 0 ;
        rem
      forever .
```

```
eq p2 =
      repeat
        'c2 := 1 ;
        while 'c1 = 1 do
          if 'turn = 1 then
            'c2 := 0 ;
            while 'turn = 1 do skip od ;
            'c2 := 1
          fi
        od ;
        crit ;
        'turn := 1 ;
        'c2 := 0 ;
        rem
```

```
        forever .

  eq initialMem = ['c1, 0] ['c2, 0] ['turn, 1] .
  eq initial = { [1, p1] | [2, p2], initialMem, 0 } .
endm
```

Model Checking Dekker's Algorithm

We need to define three state predicates parameterized by the process id: `enterCrit`, when the process is about to enter its critical section, `in-rem`, when the process is executing its remaining code fragment, and `exec`, when the process has just executed.

```
mod CHECK is inc DEKKER .  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .  *** optional
  subsort MachineState < State .
  ops enterCrit in-rem exec : Pid -> Prop .
  var M : Memory .
  vars R : Program .
  var S : Soup .
  vars I J : Pid .
  eq {[I, crit ; R] | S, M, J} |= enterCrit(I) = true .
  eq {[I, rem ; R] | S, M, J} |= in-rem(I) = true .
  eq {S, M, J} |= exec(J) = true .
```

```
endm
```

Model Checking Dekker's Algorithm (II)

The *mutual exclusion property* is satisfied:

```
reduce in CHECK : modelCheck(initial,[]~ (enterCrit(1) /\ enterCrit(2)))
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1714 in 50ms cpu (50ms real) (34280 rewrites/second)
result Bool: true
```

Model Checking Dekker's Algorithm (III)

But the *strong liveness property* that executing infinitely often implies entering one's critical section infinitely often fails, as witnessed by the counterexample,

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 16 system states.
rewrites: 159 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult:
counterexample({{[1,repeat 'c1 := 1 ; while 'c2 = 1 do
    if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ;
    crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat 'c2 := 1 ; while
    'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 :=
    1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,0] ['c2,0] [
    'turn,1],0},unlabeled}
   ...
```

Model Checking Dekker's Algorithm (IV)

Even the *weaker liveness property* that if *both* p1 and p2 execute infinitely often then both enter their critical sections infinitely often fails, due to possible looping in the rem part:

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) /\ []<> exec(2) -> []<
    /\ []<> enterCrit(2)) .
ModelChecker: Property automaton has 7 states.
ModelCheckerSymbol: Examined 236 system states.
rewrites: 1972 in 50ms cpu (50ms real) (39440 rewrites/second)
result ModelCheckResult:
counterexample({{[1,repeat 'c1 := 1 ; while 'c2 = 1 do
    if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 f
    crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat 'c2 := 1 ; w
    'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
    1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,0] ['c2,0
```

```
'turn,1],0},unlabeled}
...
```

Model Checking Dekker's Algorithm (V)

However, the *more subtle* weak liveness property that if p1 and p2 both get to execute infinitely often, then if p1 is infinitely often out of its "rem" section, then p1 enters its critical section infinitely often holds; of course, the same holds for p2.

```
reduce in CHECK : modelCheck(initial,[]<> exec(1) /\ []<> exec(2) -> []<
    -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 5 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 2219 in 60ms cpu (70ms real) (36983 rewrites/second)
result Bool: true
```

The Thread Game

A simple, yet interesting, program that we can also implement in `PARALLEL` is a "game," suggested by J Moore, between two forever-looping processes accessing a shared variable `'c` that initially holds the value 1.

Each process loop reads twice the value of `'c` in two different local variables, and then writes the sum of those two local variables back into `'c`. There is *no synchronization at all* between the processes.

Two interesting questions are: (1) which values can `'c` hold, depending on the different strategies in this game? and (2) which values can `'c` hold if *only one of the processes* is actually running?

The Thread Game (II)

The code for these processes and the relevant initial states can be defined as follows,

```
mod THREAD-GAME is
  inc PARALLEL .
  ops p1 p2 : -> Program .
  ops init init1 init2 : -> MachineState .

  eq p1 =
       repeat
         'a1 := 'c ;
         'b1 := 'c ;
         'c := 'a1 +' 'b1
       forever .
```

```
  eq p2 =
        repeat
          'a2 := 'c ;
          'b2 := 'c ;
          'c := 'a2 +' 'b2
        forever .

  eq init = { [1, p1] | [2, p2], ['c, 1], 0 } .
  eq init1 = { [1, p1], ['c, 1], 0 } .
  eq init2 = { [2, p2], ['c, 1], 0 } .
endm
```

The Thread Game (II)

We can use the `search` command in Maude to gain some experimental evidence about the first question,

```
Maude> search [1] init =>* { S:Soup, ['c, 1] M:Memory, J:Pid } .
Solution 1 (state 0)
states: 1  rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> none
J:Pid --> 0

Maude> search [1] init =>* { S:Soup, ['c, 2] M:Memory, J:Pid } .
Solution 1 (state 13)
states: 14  rewrites: 38 in 10ms cpu (10ms real) (3800 rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
```

```
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,1] ['b1,1]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 3] M:Memory, J:Pid } .
Solution 1 (state 69)
states: 70  rewrites: 326 in 0ms cpu (0ms real) (~ rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,1] ['b1,1] ['a2,1] ['b2,2]
J:Pid --> 2

Maude> search [1] init =>* { S:Soup, ['c, 4] M:Memory, J:Pid } .
search [1] in THREAD-GAME : init =>* {S:Soup,M:Memory ['c,4],J:Pid} .
states: 62  rewrites: 282 in 10ms cpu (10ms real) (28200 rewrites/second
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,2]
```

```
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 5] M:Memory, J:Pid } .
Solution 1 (state 275)
states: 276  rewrites: 1437 in 30ms cpu (30ms real) (47900 rewrites/seco
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,3] ['a2,1] ['b2,2]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 6] M:Memory, J:Pid } .
Solution 1 (state 243)
states: 244  rewrites: 1278 in 20ms cpu (20ms real) (63900 rewrites/seco
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,2] ['b1,2] ['a2,2] ['b2,4]
J:Pid --> 2
```

```
Maude> search [1] init =>* { S:Soup, ['c, 7] M:Memory, J:Pid } .
Solution 1 (state 912)
states: 913  rewrites: 4998 in 100ms cpu (100ms real) (49980 rewrites/se
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,4] ['b1,3] ['a2,1] ['b2,2]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 8] M:Memory, J:Pid } .
search [1] in THREAD-GAME : init =>* {S:Soup,M:Memory ['c,8],J:Pid} .
states: 236  rewrites: 1234 in 30ms cpu (30ms real) (41133 rewrites/seco
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,4] ['b1,4]
J:Pid --> 1

Maude> search [1] init =>* { S:Soup, ['c, 9] M:Memory, J:Pid } .
Solution 1 (state 883)
```

```
states: 884  rewrites: 4846 in 90ms cpu (90ms real) (53844 rewrites/seco
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,3] ['b1,3] ['a2,3] ['b2,6]
J:Pid --> 2

Maude> search [1] init =>* { S:Soup, ['c, 10] M:Memory, J:Pid } .
Solution 1 (state 829)
states: 830  rewrites: 4511 in 90ms cpu (90ms real) (50122 rewrites/seco
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,4] ['b1,6] ['a2,2] ['b2,4]
J:Pid --> 1

...

Maude> search [1] init =>* { S:Soup, ['c, 99] M:Memory, J:Pid } .
Solution 1 (state 68974)
```

```
states: 68975  rewrites: 408394 in 8960ms cpu (9020ms real) (45579
    rewrites/second)
S:Soup --> [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever]
    repeat 'a2 := 'c ; 'b2 := 'c ; 'c := ('a2 +' 'b2) forever]
M:Memory --> ['a1,48] ['b1,51] ['a2,3] ['b2,48]
J:Pid --> 1
```

The Thread Game (III)

We can likewise use the `rewite` command in Maude to gain some experimental evidence about the second question,

```
Maude> rewrite [20] in THREAD-GAME : init1 .
rewrite [20] in THREAD-GAME : init1 .
--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 :=
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['c,1],1}

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := '
    := ('a1 +' 'b1) forever],['c,1] ['a1,eval('c, ['c,1])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('
```

```
    'b1) forever],(['a1,1] ['c,1]) ['b1,eval('c, ['a1,1] ['c,1])],1}

--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(
    ['b1,1]) ['c,eval('a1 +' 'b1, (['a1,1] ['b1,1]) ['c,1])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 :=
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,1] ['c,2] ['b1,1

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := '
    := ('a1 +' 'b1) forever],(['c,2] ['b1,1]) ['a1,eval('c, (['c,2] ['b1
    'a1,1])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('
    'b1) forever],(['a1,2] ['c,2]) ['b1,eval('c, (['a1,2] ['c,2]) ['b1,1
```

```
--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(
    ['b1,2]) ['c,eval('a1 +' 'b1, (['a1,2] ['b1,2]) ['c,2])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 :=
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,2] ['c,4] ['b1,2

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := '
    := ('a1 +' 'b1) forever],(['c,4] ['b1,2]) ['a1,eval('c, (['c,4] ['b1
    'a1,2])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('
    'b1) forever],(['a1,4] ['c,4]) ['b1,eval('c, (['a1,4] ['c,4]) ['b1,2
```

```
--->
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(
    ['b1,4]) ['c,eval('a1 +' 'b1, (['a1,4] ['b1,4]) ['c,4])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 :=
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,4] ['c,8] ['b1,4

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := '
    := ('a1 +' 'b1) forever],(['c,8] ['b1,4]) ['a1,eval('c, (['c,8] ['b1
    'a1,4])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('
    'b1) forever],(['a1,8] ['c,8]) ['b1,eval('c, (['a1,8] ['c,8]) ['b1,4

--->
```

```
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(
    ['b1,8]) ['c,eval('a1 +' 'b1, (['a1,8] ['b1,8]) ['c,8])],1}

--->
{empty | [1,('a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1)) ; repeat 'a1 :=
    'b1 := 'c ; 'c := ('a1 +' 'b1) forever ; skip],['a1,8] ['c,16] ['b1,

--->
{empty | [1,'b1 := 'c ; 'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := '
    := ('a1 +' 'b1) forever],(['c,16] ['b1,8]) ['a1,eval('c, (['c,16] ['
    ['a1,8])],1}

--->
{empty | [1,'c := ('a1 +' 'b1) ; repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('
    'b1) forever],(['a1,16] ['c,16]) ['b1,eval('c, (['a1,16] ['c,16]) ['
    8])],1}

--->
```

```
{empty | [1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b1) forever],(
    16] ['b1,16]) ['c,eval('a1 +' 'b1, (['a1,16] ['b1,16]) ['c,16])],1}

result MachineState: {[1,repeat 'a1 := 'c ; 'b1 := 'c ; 'c := ('a1 +' 'b
    forever],['a1,16] ['c,32] ['b1,16],1}
Maude>
```

The Thread Game (IV)

The above experimental evidence suggests the following two *conjectures*:

1. when both processes are running, then for any $n \geq 1$ there is an execution such that `'c` eventually holds n
2. when only one process is running, then `'c` will initially hold 1, and then for each $n \geq 0$ if it holds 2^n, it will continue holding that value until it eventually holds 2^{n+1}.

Can you prove it? (**Note**: Any precise mathematical proof will do; do not even need to use temporal logic).

A Semantic Framework for Programming Languages

`PARALLEL` is a toy language. Can the rewriting logic approach *scale up* to real concurrent languages? The answer is "yes." We can define the semantics of a concurrent programming language L by a rewrite theory $\mathcal{R}_L = (\Sigma_L, E_L, R_L)$, where:

- Σ_L specifies L's *syntax* and the auxiliary operators needed in semantic definitions (memory, environment, etc.)
- the equations E_L specify the semantics of all the *deterministic features* of L and of the auxiliary semantic operations.
- the rewrite rules R_L specify the semantics of all the *concurrent features* of L.

Execution and Formal Analysis of Concurrent Programs

Once a definition of a language is given in Maude, we get an interpreter for free and we also get:

1. a semi-decision procedure to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude's `search` command;
2. an LTL model checker for finite-state programs or program abstractions;
3. a theorem prover (Maude's ITP) that can be used to semi-automatically prove programs correct.

Specifying Java and JVM

Java has been recently defined at UIUC by Feng Chen, using a CPS semantics as above, with 600 equations and 15 rewrite rules. Azadeh Farzan has developed a more direct specification for the JVM, not based on continuations, with around 300 equations and 40 rewrite rules.

Both the Java and the JVM specifications include multithreading, inheritance, polymorphism, object references, and dynamic object allocation. Native methods and most Java libraries are not supported at present.

JavaFAN Project

Based on Maude rewriting logic specifications of Java and JVM, we are developing JavaFAN (Java Formal ANalyzer), a tool in which Java and JVM code can be executed and analyzed. The following figure shows the architecture of JavaFAN.

Performance of JavaFAN

Tests	**JVM**	**Java**	**Other**
Remote Agent (s)	0.3	0.1	2 (Stanford)
2-stage Pipeline	17m	—	100m+ (Stanford)
DinPhil (4)	0.64	1.2	—
DinPhil (6)	33.3	81.7	—
DinPhil (8)	13.7m	98m	—
DinPhil (9)	803.2m	—	—
Deadlock-free DinPhil (5)	3.2m	19.2	∞ (JPF)
Deadlock-free DinPhil (7)	686.4m	27m	∞ (JPF)
Thread Game (100) (s)	17.1	6.6	—
Thread Game (1000) (s)	10.1m	5.1m	—

Performance of JavaFAN: Some discussion

There are essentially two reasons for JavaFAN to compare favorably with more conventional Java analysis tools: (1) the high performance of Maude for execution, search, and model checking; and (2) optimized equational and rule definitions.

The second reason is the use of performance-enhancing specification techniques at the Maude level, including:

- expressing as equations E the semantics of all deterministic computations, and as rules R only concurrent computations.
- favoring *unconditional* equations and rules over less efficient conditional versions.
- using a continuation passing style in semantic equations.

Other Language Case Studies

Similar positive experience in using rewriting logic and Maude to give semantics definitions of concurrent programming languages and getting interpreters and program analysis tools for free for those languages is reported in several papers, including the surveys by Meseguer and Roşu in: (i) Proc. IJCAR'04, Springer LNCS 3097; and (ii) Proc. SOS'05, Elsevier ENTCS.

In particular, semantic definitions have already been given in Maude for substantial subsets of the following languages: ABEL, bc, Beta, CCS, CIAO, CML, Creol, ELOTOS, Haskell, Lisp, LLVM, MSR, Pi-Calculus, Pict, PLAN, Python, Ruby, SIMPLE, and Samalltalk.

Model Checking Safety Properties of Infinite-State Syste

As usual for model checkers, the Maude LTL model checker relies quite essentially on the assumption that the set of states reachable from an initial state is *finite*. Yet, many concurrent systems of interest fail to have this property.

For example, just adding some simple fault-tolerant features to a simple asynchronous communication protocol can easily bring us into an infinity of reachable states. Likewise, cypto protocols with powerful enough intruder models easily have infinite sets of reachable states.

What can we do in such cases? Is some kind of model checking still applicable?

Model Checking Safety Properties of Infinite-State Syste

In practice, the most vital properties are often *safety properties*. For such properties, under reasonable assumptions, we can indeed do something quite practical, namely *breadth-first search for a counterexample*.

This provides a *semidecision procedure* for the *failure* of the safety property. That is, if the safety property fails we will always *find this out after a finite number of steps* (in practice, of course, should be "finite enough" so as not to run out of memory!).

Because of the infinite search involved, by this method *we can never know for sure* that the safety property holds; but, disregarding time and space limitations, we have a *complete method* for *finding safety bugs* in a specification or program.

Infinite-State Model Checking in Practice

But we have to be more precise. What are the "reasonable asumptions" about the rewrite theory $\mathcal{R}$ and the safety poperties to be checked?

As usual, $\mathcal{R} = (\Sigma, E, \phi, R)$ should have (Σ, E) confluent, terminating and sort-decreasing. And the rules R should be coherent relative to E, have no extra variables (except "matching" ones) in their righthand sides, and have equational conditions. These are the standard assumptions for the executability of Maude system modules.

Regarding the safety property, we assume properties of the form $P \rightarrow \Box Q$, with P and Q *decidable state predicates* such that:

Infinite-State Model Checking in Practice (II)

- we can effectively determine that the set of states satisfying P is *finite*, and we can enumerate those states, say, $\{init_1, \ldots, init_n\}$
- Q can be assumed to be an *arbitrary Boolean combination* of unparameterized state predicates $p_1, \ldots, p_m$, each of which is defined *by the general method already explained*, that is, by extending $\mathcal{R}$ with equations,

```
ceq pi(exp1) = true if C1 .
...
ceq pi(expk) = true if Ck .
```

Infinite-State Model Checking in Practice (III)

By De Morgan's laws the *negation* $\neg Q$ is also a Boolean combination of the form $\neg Q(S) = bexp(p_1(S), \ldots, p_m(S))$, which we can assume in *negative normal form*.

In Maude we can express each basic predicate appearing in negated form in the negative normal form Boolean expression $\neg Q(S) = bexp(p_1(S), \ldots, p_m(S))$, say $\neg p_j(S)$, by `pj(S)=/= true`.

Infinite-State Model Checking in Practice (IV)

Under the above assumptions about $\mathcal{R}$, P, and Q, our desired semidecision procedure for the failure of the safety property $P \rightarrow \Box Q$ can be implemented in Maude by running in parallel, or in a fair scheduling of n processes, the n (breadth-first) search commands, $1 \leq j \leq n$,

`search [1]` $init_j$ `=>*` `S` `s.t.` $bexp(p_1(S), \ldots, p_m(S))$.

Of course, this is not the *only method* of reasoning about infinite-state systems. Two other alternative methods are: (1) *deductive proof*; and (2) *model checking an abstraction*, that is, associating to $\mathcal{R}$ a more abstract version $\mathcal{A}$ whose sets of reachable states are finite and such that

$$T_{Reach(\mathcal{A})} \models_{LTL} \varphi \;\Rightarrow\; T_{Reach(\mathcal{R})} \models_{LTL} \varphi.$$

A Cryptographic Protocol Example

We can illustrate the power of this model checking technique for safety properties of infinite state systems by showing how it can be used to find subtle *attacks* for *cryptographic protocols*, including some that have been used extensively and have been considered secure for a long time.

One such protocol is the 1978 Needham-Schroeder authentication protocol (NSPK) for which a subtle "man-in-the-middle" attack was found by G. Lowe in 1996 using model checking.

The goal of the NSPK Protocol is to provide *authentication* of two agents who want to be assured of each other's identity before they exchange safety-critical data.

A Cryptographic Protocol Example (II)

That is, an intruder should not be allowed to impersonate another agent. For this purpose, initiator and responder of a communication mutually authenticate each other.

NSPK uses *public key cryptography*, i.e., each agent has a public key which can be accessed by all agents, and a secret key which is the inverse of the public key.

Moreover, *nonces* are used in the protocol. Nonces are freshly generated, unguessable random numbers to be used in a single run of the protocol.

A Cryptographic Protocol Example (III)

Here is a textbook-style simplified description of NSPK:

Message 1	$A \rightarrow B : A.B.\{N_a, A\}_{PK(B)}$
Message 2	$B \rightarrow A : B.A.\{N_a.N_b\}_{PK(A)}$
Message 3	$A \rightarrow B : A.B.\{N_b\}_{PK(B)}$

This level of description is *ambiguous*, in that a fair amount of *implicit assumptions* are *left unspecified*. An object-oriented rewriting logic specification of the protocol (developed in joint work with G. Denker and C. Talcott) makes these assumptions explicit, and allows model checking.

Maude Specification of NSPK

We first specify key *algebraic properties* of the *cryptographic infrastructure* in a functional module.

```
fmod DATATYPES is
  sorts Key Field Nonce Principal Run Role EstabComm .
  subsort Nonce Principal Key < Field .
  op keypair : Key Key -> Bool [comm] .
  op ped : Key Field -> Field . *** encryption function
  op n : Principal Nat -> Nonce .
  ceq ped(sk,ped(pk,f)) = f  if keypair(sk,pk) .
  ...
endfm
```

The protocol itself, as well as the actions of an attacker, are specified as follows (fragment):

Maude Specification of NSPK (II)

```
class Agent | e_com: EstabCom, sec_key: Key, role_i: Run, role_r:
              d_com: FieldSet cnt: Nat .

msg from_to_send_ : Principal Principal Field  -> Message .

vars A B P : Principal . vars RI RR : Run . vars NI : Nonce . ...
rl [BeginRun] :
  < A : Agent | role_i: RI, d_com: B U S, cnt: J >
  => < A : Agent | role_i: RI U (n(A,J),B,mtfield), d_com: S, cnt
     from(A)to(B)send(ped(pk(B),n(A,J),A)) .

crl [Message1Rec] :
  < B : Agent | sec_key: SKB, role_i: RI, role_r: RR, cnt: J >
```

```
from(A)to(B)send(ped(PKB,F,A))
=> < B : Agent | role_r: RR U (n(B,J),A,F), cnt: J + 1 >
   from(B)to(A)send(ped(pk(A),F.n(B,J)))
if keypair(SKB,PKB) and not(F in RR) .
```

Maude Specification of NSPK (III)

```
crl [Message2RecCorrect] :
  < A : Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield), e_com:
  from(B)to(A)send(ped(PKA,F))
  => < A : Agent | role_i: RI, ECom: C U (i,NI,B,rest(F)) >
     from(A)to(B)send(ped(pk(B),rest(F)))
  if keypair(SKA,PKA) and (B == P) and (NI == first(F)) .

crl [Message2RecIncorrect] :
  < A : Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield) >
  from(B)to(A)send(ped(PKA,F))
  => < A : Agent | role_i: RI >
  if keypair(SKA, PKA) and (NI == first(F)) and (B =/= P) .
```

Maude Specification of the Intruder

```
class Intruder | e_com: EstabCom sec_key: Key ncs: FieldSet,
                 msgs: FieldSet agents: FieldSet role_i: Run,
                 role_r: Run d_com: Field cnt: Nat.
```

```
crl [IntruderFakeMessage] :
  < I : Intruder | ncs: N U F, agents: S U A U B >
  => < I : Intruder | ncs: N U F > from(A)to(B)send(ped(pk(B),F))
  if B =/= I .
```

2 similar: IntruderInterceptMessage, IntruderOverhearMessage, IntruderReplayMessage

The State Predicate of an Attack

In a topmost version of the specification, the situation where *authentication information has been compromised* is specified by the following state predicate:

```
op attack? : Configuration -> Prop .

ceq attack?(boundary(
     < INTR : Intruder | ecom : EC, rolei : RI, roler : RR,
                         ncs : fset+(fset+(FSET1, N1), N2) >
     < A : NSPKAgent | ecom : ecom+(EC2, ecom(ROLE,N1,B,N2)) >
     Conf)) = true
     if (not(inEstabCom(ecom(r,N2,A,N1),EC))
         and not(inEstabCom(ecom(i,N2,A,N1), EC))
         and not(in(N1,RI)) and not(in(N1,RR))
         and not(in(N2,RI)) and not(in(N2,RR))
         and B =/= INTR) == true .
```

Finding an Attack

The relevant *safety property* is that no such attack is possible under reasonable initial conditions. For example, we can consider a simple scenario with two agents and an attacker given by an initial state `cf2Agents1Intruder` equationally defined in the obvious manner. Then the desired safety property is $P \rightarrow \Box Q$ with:

$P = (S = \texttt{cf2Agents1Intruder})$

$Q = \neg(\texttt{attack?}(S) = \texttt{true})$.

Maude's `search` command finds Lowe's countarexample to such a property:

Finding an Attack (II)

```
Maude> search [1] cf2Agents1Intruder =>+ C:Configuration s.t.
                                        attack?(C:Configuration) = true .
search [1] in NSPK : cf2Agents1Intruder =>+ C:Configuration such that
                                        attack?(C:Configuration) = true .
Solution 1 (state 37826)
states: 37827 in 25350ms cpu (44300ms real)
C:Configuration -->
 boundary(< alice : NSPKAgent | cnt : 2,dcom : mtfset,roler
 : mtrun,rolei : mtrun,seckey : skalice,ecom : ecom(i, n(alice, 1), mrx,
 bob, 1)) > < bob : NSPKAgent | cnt : 2,dcom : mtfield,roler : mtrun,rol
 mtrun,seckey : skbob,ecom : ecom(r, n(bob, 1), alice, n(alice, 1)) > <
 : Intruder | cnt : 1,dcom : mtfield,roler : mtrun,rolei : mtrun,seckey
 skmrx,ecom : mtecom,agents : fset+(alice, bob, mrx),ncs : fset+(mtfset,
 alice, 1), n(bob, 1)),msgs : fset+(mtfset, ped(pkalice, cat(n(alice, 1)
 bob, 1)))) >)
```

Finding an Attack (III)

We can find the actual sequence of rewrites leading to the attack by giving to Maude the command `show path 37826 .` A detailed trace is then shown, corresponding to the sequence of rewrite rule applications:

```
[BeginRun];[IntruderAcceptEveryMessage1];[IntruderFakeMessage1];

[Message1Rec];[IntruderInterceptMessage2];[IntruderReplayMessage];

[Message2Rec];[IntruderAcceptEveryMessage3];

[IntruderFakeMessage3];[Message3Rec]
```

Simulations

Given two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$, both having the same set AP of atomic propositions, an AP-*simulation* $H : \mathcal{A} \longrightarrow \mathcal{B}$ of $\mathcal{A}$ by $\mathcal{B}$ is a *total* binary relation $H \subseteq A \times B$ such that, denoting pairs $(a, b) \in H$ by aHb, we have:

- if $a \rightarrow_{\mathcal{A}} a'$ and aHb, then there is a $b' \in B$ such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, and
- $(\forall\, a \in A)(\forall\, b \in B)\ \ aHb \ \ \Rightarrow \ \ L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

If the relation H is a *function*, then we call H an AP-*simulation map*. If both H and H^{-1} are AP-simulations, then we call H a *bisimulation*.

Simulations (II)

Note that (exercise) AP-simulations (resp. AP-simulation maps, resp. AP-bisimulations) *compose*. That is, if we have AP-simulations (resp. AP-simulation maps, resp. AP-bisimulations)

$$H : \mathcal{A} \longrightarrow \mathcal{B} \qquad G : \mathcal{B} \longrightarrow \mathcal{C}$$

then $H; G : \mathcal{A} \longrightarrow \mathcal{C}$ is also an AP-simulation (resp. AP-simulation map, resp. AP-bisimulation).

Note also that the identity function 1_A is trivially an AP-simulation $1_A : \mathcal{A} \longrightarrow \mathcal{A}$, and also an AP-simulation map and an AP-bisimulation.

$LTL^{-}(AP)$

It is sometimes useful to restict ourselves to a *negation-free* fragment $LTL^{-}(AP)$ of $LTL(AP)$, defined as follows:

- **Atomic Propositions**. If $p \in AP$, then $p \in LTL^{-}(AP)$.
- **Next Operator**. If $\varphi \in LTL(AP)$, then $\bigcirc\varphi \in LTL(AP)$.
- **Until and Release Operators**. If $\varphi, \psi \in LTL^{-}(AP)$, then $\varphi \,\mathcal{U}\, \psi \in LTL^{-}(AP)$, and $\varphi \,\mathcal{R}\, \psi \in LTL^{-}(AP)$.
- **Boolean Connectives**. $\top, \bot \in LTL^{-}(AP)$. If $\varphi, \psi \in LTL^{-}(AP)$, then $\varphi \vee \psi \in LTL^{-}(AP)$, and $\varphi \wedge \psi \in LTL^{-}(AP)$.

$LTL^-(AP)$ **(II)**

Since $LTL^-(AP)$ is a sublogic of $LTL(AP)$, its semantics is the same. Furthermore, in a very practical sense *there is no real loss of generality* by resticting ourselves to formulas in $LTL^-(AP)$, because we can always transform *any* LTL formula φ into a *semantically equivalent* LTL^- formula $\hat{\varphi}$.

The idea is as follows. Consider a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ on AP, and any formula $\varphi \in LTL(AP)$, and let $nnf(\varphi)$ be its *negative normal form* (Cf. Lecture 23) in which all negations have been pushed next to the atoms. Of course we have,

$$\mathcal{A} \models \varphi \quad \Leftrightarrow \quad \mathcal{A} \models nnf(\varphi).$$

$LTL^{-}(AP)$ **(III)**

Consider the extended set of atomic propositions $\hat{AP} = AP \cup \overline{AP}$, where $\overline{AP} = \{\overline{p} \mid p \in AP\}$. Then define the $LTL^{-}(\hat{AP})$ formula $\hat{\varphi}$ as the result of replacing each negated atom $\neg p$ in $nnf(\varphi)$ by $\overline{p}$, and otherwise leaving the rest of $nnf(\varphi)$ unchanged.

We can then extend $\mathcal{A}$ to a Kripke structure $\hat{\mathcal{A}}$ on $\hat{AP}$ as follows.
$\hat{\mathcal{A}} = (A, \rightarrow_{\mathcal{A}}, L_{\hat{\mathcal{A}}})$, where for each $a \in A$,
$L_{\hat{\mathcal{A}}}(a) = L_{\mathcal{A}}(a) \cup \{\overline{p} \mid p \notin L_{\mathcal{A}}(a)\}$. Then we have (exercise)

$$\mathcal{A} \models \varphi \quad \Leftrightarrow \quad \hat{\mathcal{A}} \models \hat{\varphi}.$$

$LTL^{-}(AP)$ (IV)

Note that, in terms of our general method for defining atomic predicates for a Maude system module and using the Maude LTL model checker, the above method of transforming formulas into negation-free ones is very easy to accomplish. For each state predicate

```
op p: S1 ... Sn -> Prop .
```

we introduce another state predicate,

```
op ~p: S1 ... Sn -> Prop .
```

and give the equation,

```
eq S:State |= ~p(X1,...,Xn) = (S:State |= p(X1,...,Xn) =/= true) .
```

To get the translation $\varphi \mapsto \hat{\varphi}$ we add to `LTL`, and apply to $nnf(\varphi)$, the *otherwise* equations (cf. Maude 2.0)

```
eq ~ p(X1,...,Xn) = ~p(X1,...,Xn) [owise] .
```

Simulations Reflect Satisfaction of LTL^- Formulae

We say that an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *reflects* the satisfaction of an LTL formula φ iff $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$.

A fundamental result, allowing us to prove the satisfaction of an LTL formula φ in an infinite-state system $\mathcal{A}$ by proving the same satisfaction in a finite-state system $\mathcal{B}$ that simulates it is the following,

Theorem: AP-simulations always reflect satisfaction of $LTL^-(AP)$ formulae.

Simulations Reflect Satisfaction of LTL^- Formulae (II)

bf Proof: First of all, note that we can extend H to a binary relation $\hat{H} : Path(\mathcal{A}) \longrightarrow Path(\mathcal{B})$, where,

$$\pi \hat{H} \rho \quad \Leftrightarrow \quad \forall\, n \in \mathbb{N}\ \ \pi(n) H \rho(n).$$

Lemma1: If aHb, than for each $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$ there is a $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$ and $\pi \hat{H} \rho$.

Proof of Lemma1: The proof amounts to an inductive argument that, if we have built the first n stages of ρ, then we can always build the $n+1$ stage. So, assume that we have ρ defined for $0 \leq i \leq n$ with $\rho(0) = b$, and with:

Simulations Reflect Satisfaction of LTL^- Formulae (III)

- $\pi(i) H \rho(i)$, $0 \leq i \leq n$, and
- $\rho(i) \rightarrow_{\mathcal{B}} \rho(i+1)$, $0 \leq i < n$.

Then, since H is a simulation, and since $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$, we can find a $b' \in B$ such that $\rho(n) \rightarrow_{\mathcal{B}} b'$, and $\pi(n+1) H b'$. Therefore, we can define $\rho(n+1) = b'$ and extend ρ to $n+1$ steps. q.e.d.

We will be essentially done if we prove the following,

Lemma2: For each aHb, $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$, $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$ and $\pi \hat{H} \rho$; and for each $\varphi \in LTL^-(AP)$ we have,

$$\mathcal{B}, b, \rho \models \varphi \quad \Rightarrow \quad \mathcal{A}, a, \pi \models \varphi.$$

Simulations Reflect Satisfaction of LTL^- Formulae (IV)

Proof of Lemma2: The proof is by structural induction on the structure of LTL^- formulae. We prove the base case and the case $\varphi = \bigcirc\psi$, and leave the rest as an exercise.

For $p \in AP$ we have, $\mathcal{B}, b, \rho \models p$ iff $p \in L_{\mathcal{B}}$, which by H simulation and aHb implies $p \in L_{\mathcal{A}}$, which is equivalent to, $\mathcal{A}, a, \pi \models \varphi$.

Assume that the result holds for ψ. Let us then show that it holds for $\varphi = \bigcirc\psi$. We have, $\mathcal{B}, b, \rho \models \bigcirc\psi$ iff $\mathcal{B}, \rho(1), s; \rho \models \psi$. But note that if $\pi\hat{H}\rho$, then for any $n \in \mathbb{N}$ we have, $s^n; \pi\hat{H}s^n; \rho$. Therefore, by the induction hypothesis we can conclude that,
$\mathcal{A}, \pi(1), s; \pi \models \psi$, which is equivalent to, $\mathcal{A}, a, \pi \models \bigcirc\psi$. q.e.d.

Simulations Reflect Satisfaction of LTL^- Formulae (V)

We are now essentially done. Suppose $\mathcal{B}, b \models \varphi$ and aHb. Then we have, $\mathcal{B}, b, \rho \models \varphi$ for any $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$. To prove $\mathcal{A}, a \models \varphi$, we have to show that for each $\pi \in Path(\mathcal{A})$ with $\pi(0) = a$ we have, $\mathcal{A}, a, \pi \models \varphi$. But, by Lemma1, for each such π we have a ρ such that, $\rho(0) = b$, and $\pi \hat{H} \rho$. Then, by Lemma2, we have, $\mathcal{A}, a, \pi \models \varphi$. q.e.d.

We say that an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *preserves* the satisfaction of an LTL formula φ iff $\mathcal{A}, a \models \varphi$ and aHb imply $\mathcal{B}, b \models \varphi$.

Corollary: AP-bisimulations always reflect and preserve satisfaction of $LTL^-(AP)$ formulae.

Exercises

Ex.24.1 Call an AP-simulation (resp. simulation map, resp. bisimulation) *strict* if aHb implies $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$. Prove the following,

Theorem: Strict AP-simulations always reflect satisfaction of $LTL(AP)$ formulae.

Ex.24.2 Prove the following,

Corollary: Strict AP-bisimulations always reflect and preserve satisfaction of $LTL(AP)$ formulae.

Abstraction Methods

To prove that a, possibly infinite-state, Kripke structure $\mathcal{A}$ and initial state a satisfy an LTL^- formula φ, with, say, AP the set of atomic propositions actually appearing in φ, it is enough to find an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ and an initial state $b \in B$ such that aHb and:

- the set of states reachable from b in $\mathcal{B}$ is finite, and
- $\mathcal{B}, b \models \varphi$.

Then, we can model check the property $\mathcal{B}, b \models \varphi$ to prove $\mathcal{A}, a \models \varphi$. Methods to find such an H are called *abstraction methods*. This lecture will describe some recent results in joint work with Narciso Martí-Oliet and Miguel Palomino on abstraction methods for systems specified by rewrite theories.

Quotient Abstractions

Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ be a Kripke structure on AP, and let $\equiv$ be an arbitrary equivalence relation on A. We can use $\equiv$ to define a new Kripke structure, $(\mathcal{A}/\equiv) = (A/\equiv, \rightarrow_{\mathcal{A}/\equiv}, L_{\mathcal{A}/\equiv})$, where:

- $[a_1] \rightarrow_{\mathcal{A}/\equiv} [a_2]$ iff $\exists\ a'_1 \in [a_1]\ \exists\ a'_2 \in [a_2]$ s.t. $a'_1 \rightarrow_{\mathcal{A}} a'_2$.
- $L_{\mathcal{A}/\equiv}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$

It is then trivial to check that the projection map to equivalence classes $q_{\equiv} : a \mapsto [a]$ is an AP-simulation map $q_{\equiv} : \mathcal{A} \longrightarrow \mathcal{A}/\equiv$, which we call the *quotient abstraction* defined by $\equiv$.

Equational Quotient Abstractions

We are of course particularly interested in abstraction methods for systems specified by rewrite theories. Recall that, given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ for which we have equationally defined state predicates Π in a kind k of states, we can associate to its initial reachability model $(T_{\Sigma/E}, \rightarrow_{\mathcal{R}})$ the Kripke structure,

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow_{\mathcal{R}}^{1})^{\bullet}, L_{\Pi})$$

Equational Quotient Abstractions (II)

It is enough to focus on those *state predicates actually occurring* in a particular formula φ, which we may assume have been defined by the general method described when explaining the Maude LTL model checker.

We assume that $\mathcal{R}$ has been defined as a Maude module, and that we have *equationally specified* with equations D a finite set of (possibly parametric) state predicate symbols Π. Then our (possibly infinite) set of atomic predicates is the set $AP_\Pi = \{\theta(p) \mid p \in \Pi,\ \theta \text{ canonical ground substitution}\}$. This defines a labeling function,

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid (E \cup D) \vdash t\,\texttt{|=}\theta(p) = \texttt{true}\}.$$

We are then interested in the Kripke structure,

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow^{1}_{\mathcal{R}})^{\bullet}, L_{\Pi}).$$

Equational Quotient Abstractions (III)

Let $\mathcal{R} = (\Sigma, E, \phi, R)$ be our rewrite theory. Then, a quite general method for defining quotient abstractions of the Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^{\bullet}, L_{\Pi})$ is by specifying an equational theory extension of the form,

$$(\Sigma, E) \subseteq (\Sigma, E \cup E').$$

Since this defines an equivalence relation $\equiv_{E'}$ on $T_{\Sigma/E,k}$, namely,

$$[t]_E \equiv_{E'} [t']_E \quad \Leftrightarrow \quad E \cup E' \vdash (\forall \emptyset)\ t = t' \quad \Leftrightarrow \quad [t]_{E \cup E'} = [t']_{E \cup E'},$$

then we can obviously define our quotient abstraction as, $\mathcal{K}(\mathcal{R}, k)_{\Pi} / \equiv_{E'}$. We call this the *equational quotient abstraction* of $\mathcal{K}(\mathcal{R}, k)_{\Pi}$ defined by E'.

Equational Quotient Abstractions (IV)

But can $\mathcal{K}(\mathcal{R},k)_\Pi/\equiv_{E'}$, which we have just defined in terms of the underlying Kripke structure $\mathcal{K}(\mathcal{R},k)_\Pi$, be understood as the Kripke structure associated to *another rewrite theory*? Let us take a closer look at,

$$\mathcal{K}(\mathcal{R},k)_\Pi/\equiv_{E'} = (T_{\Sigma/E,k}/\equiv_{E'}, (\rightarrow^1_{\mathcal{R}})^\bullet/\equiv_{E'}, L_{\Pi/\equiv_{E'}}).$$

The first obvious observation is that, by definition, we have, $T_{\Sigma/E,k}/\equiv_{E'} \cong T_{\Sigma/E\cup E',k}$. A second, similarly obvious, observation (exercise) is that if $\mathcal{R}$ is k-*deadlock-free*, that is, if we have, $(\rightarrow^1_{\mathcal{R}})^\bullet = (\rightarrow^1_{\mathcal{R}})$, then the rewrite theory $\mathcal{R}/E' = (\Sigma, E\cup E', \phi, R)$ is also k-deadlock-free, and we have,

$$(\rightarrow^1_{\mathcal{R}/E'})^\bullet = (\rightarrow^1_{\mathcal{R}/E'}) = (\rightarrow^1_{\mathcal{R}})^\bullet/\equiv_{E'}\ .$$

Equational Quotient Abstractions (V)

Therefore, for $\mathcal{R}$ k-deadlock-free, our obvious candidate for a rewrite theory having $\mathcal{K}(\mathcal{R},k)_{\Pi}/\equiv_{E'}$ as its underlying Kripke structure is the rewrite theory, $\mathcal{R}/E' = (\Sigma, E \cup E', \phi, R)$, that is, we just add to $\mathcal{R}$ the equations E', and *do not change at all* the rules R.

How restrictive is the requirement that $\mathcal{R}$ is k-deadlock-free? There is *no real loss of generality*. To a theory $\mathcal{R}$ satisfying the usual executability assumptions and having equational conditions we can always associate a *semantically equivalent* (from the LTL point of view) theory $\mathcal{R}^k_{d.f.}$ which is k-deadlock-free (see **Ex**.25.3).

Therefore, at a purely *mathematical* level, $\mathcal{R}/E'$ seems to be what we want. The problem comes with the following two *executability questions*:

Equational Quotient Abstractions (VI)

- is the theory $(\Sigma, E \cup E')$ *ground confluent, sort-decreasing and terminatig*?
- are the rules R *ground coherent* relative to $E \cup E'$?

The answer to each of these questions may be positive or negative. Any Maude user worth his or her salt *should* be able to specify the E' so that the first question is answered in the affirmative; but in any case we can always attempt to *check* such a property with Maude tools; and if the check fails we can even try to *complete* the equations with the KB tool, so that we get a theory (Σ, E'') equivalent to $(\Sigma, E \cup E')$ for which the first question has an affirmative answer.

Equational Quotient Abstractions (VII)

Likewise, we can try to check whether the rules R are ground coherent relative to $E \cup E'$ (or to E'') and if the check fails we can again try to *complete* the rules R to a semantically equivalent set of rules R' (no suitable Maude tools exist for this at the moment; they are in their design stage).

By this process we can hopefully arrive at an *executable* theory $\mathcal{R}' = (\Sigma, E'', \phi, R')$ which is *semantically equivalent* to $\mathcal{R}/E'$. We can then use $\mathcal{R}'$ to try to model check properties about $\mathcal{R}$.

But we are not finished yet. What about the state predicates Π?

Equational Quotient Abstractions (VIII)

Recall that these (possibly parameterized) state predicates will have been defined by means of equations D in a Maude module importing the specification of $\mathcal{R}$ and also the `MODEL-CHECKER` module. The question is whether the state predicates Π are *preserved* under the equations E'.

This indeed may be a problem. We need to unpack a little the definition of the innocent-looking labeling function, $L_{\Pi/\equiv_{E'}}$, which is of course defined by the intersection formula,

$$L_{\Pi/\equiv_{E'}}([t]_{E \cup E'}) = \bigcap_{[x]_E \subseteq [t]_{E \cup E'}} L_{\Pi}([x]_E).$$

Equational Quotient Abstractions (IX)

In general, computing such an intersection and coming up with new equational definitions D' capturing the new labeling function $L_{\Pi/\equiv_{E'}}$ may not be easy. It becomes much easier if the state predicates Π are preserved under the equations E'.

By definition, we say that the state predicates Π are *preserved* under the equations E' if for any $[x]_E, [y]_E \in T_{\Sigma/E,k}$ we have the implication,

$$[x]_{E \cup E'} = [y]_{E \cup E'} \quad \Rightarrow \quad L_\Pi([x]_E) = L_\Pi([y]_E).$$

Equational Quotient Abstractions (X)

Note that, in that case, and assuming that the equations $E \cup E' \cup D$ (or $E'' \cup D$) are ground confluent, sort-decreasing, and terminating, we *do not need to change the equations* D to define the state predicates Π on $\mathcal{R}/E'$ (or its semantic equivalent $\mathcal{R}'$).

Therefore, we have an isomorphism (given by a pair of invertible bisimulation maps)

$$\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi,$$

or, in case we need the semantically-equivalent $\mathcal{R}'$, an isomorphism,

$$\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'} \cong \mathcal{K}(\mathcal{R}', k)_\Pi.$$

Equational Quotient Abstractions (XI)

The crucial point in both isomorphisms is that the labeling fuction of the righthand side Kripke structure is now *equationally defined* by the same equations D as for the extension of $\mathcal{R}$. So we can do it all in Maude.

Furthermore, the quotient AP_Π-simulation map,

$$\mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/E', k)_\Pi$$

is then by construction *strict*, and therefore reflects satisfaction of *arbitrary* LTL formulae (cf. **Ex**.24.1).

Equational Quotient Abstractions (XII)

How can we actually try to *prove* the implication

$$[x]_{E \cup E'} = [y]_{E \cup E'} \quad \Rightarrow \quad L_\Pi([x]_E) = L_\Pi([y]_E)$$

to show the desired preservation of state predicates?

A particularly easy case is that of k-*topmost* rewrite theories. By definition these are rewrite theories in which all nonvariable terms t in the kind k of states are such that each variable $x \in vars(t)$ must necessarily have a kind, say k', with $k' \neq k$.

Equational Quotient Abstractions (XIII)

Suppose a k-topmost rewrite theory in which all equations in E' are of the form,

$$t = t'$$

with t and t' of kind k. Suppose, furthermore, that the equations $E \cup E' \cup D$ are ground confluent, sort-decreasing, and terminating. Then, it is relatively easy to prove as an exercise (**Ex**.25.4.) that if for each such equation and each state predicate p we can prove the inductive property,

$$E \cup D \vdash_{ind} \ (\forall \vec{x}\ \forall \vec{y})\ (t(\vec{x})\,\texttt{|=}\,p(\vec{y}) = \texttt{true} \ \Leftrightarrow \ t'(\vec{x})\,\texttt{|=}\,p(\vec{y}) = \texttt{true})$$

then we have established the preservation of the state predicates Π by the equations E'. We can use a tool like Maude's ITP to mechanically discharge proof obligations of this kind.

Equational Quotient Abstractions (XIV)

The above method of proving properties based on equational quotient abstractions can be used beyond the case of purely *propositional* LTL properties. In some cases we can also prove LTL properties involving free variables as parameters.

Let $\mathcal{R} = (\Sigma, E, \phi, R)$ be a rewite theory, and let $\varphi \in LTL(\Sigma', E \cup D)$ be a formula with *free variables*, say $x_1 : s_1, \ldots, x_n : s_n$, defined in a protecting extension of (Σ, E) and involving a finite set of possibly parametric state predicates Π which we assume defined in a computable way by $E \cup D$ ground confluent, sort-decreasing and terminating.

Equational Quotient Abstractions (XV)

Suppose, further, a formula $Q \in FOL(\Sigma, E)$ whose variables are among those in $x_1 : s_1, \ldots, x_n : s_n$, and that characterizes a set of initial states for which we want to prove,

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} \models_{LTL} Q \rightarrow \varphi.$$

Assume, finally, that we have been able to define an equational abstraction $\mathcal{R}' = (\Sigma, E'', \phi, R')$ of $\mathcal{R}$ satisfying all the requirements discussed so far (in particular the simulation map is strict) and such that:

- for each $1 \leq i \leq n$, $T_{\Sigma/E'', s_i}$ is a *finite* set; there is therefore a *finite* set of E''-canonical substitutions $\theta_1, \ldots \theta_m$ instantiating the variables $x_1 : s_1, \ldots, x_n : s_n$ in $T_{\Sigma/E''}$.

Equational Quotient Abstractions (XVI)

- likewise, for k the kind of states, and for each $1 \leq j \leq m$, the set of E''-canonical ground terms $t_{j,l}$ of kind k such that $E'' \vdash (\forall \emptyset)\ \theta_j[S \mapsto t_{j,l}](Q)$ is likewise *finite*, with $1 \leq l \leq q_j$, and can be effectively determined (that is, on $T_{\Sigma/E''}\ Q$ defines a *finite* set of initial states)

Then, we have the following useful result, which is left as an exercise:

Theorem: If for each j, $1 \leq j \leq m$, and for each l, $1 \leq l \leq q_j$ we can prove by model checking that,

$$\mathcal{K}(\mathcal{R}', k)_{\Pi}, [t_{j,l}] \models_{LTL} \theta_j(\varphi),$$

then we have proved,

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} \models_{LTL} Q \rightarrow \varphi.$$

A Bakery Protocol Example

In the early stages of algebraic specification all formalisms would use a `STACK` data type as an example. In the relatively early stages of abstraction techniques it is likewise *de rigueur* to show how the technique in question handles Lamport's bakery protocol.

This is an infinite-state protocol that achieves mutual exclusion between processes by the usual method common in bakeries and deli shops: there is a number dispenser, and customers are served in sequential order according to the number that they hold.

A simple Maude specification for the case of two processes is as follows,

A Bakery Protocol Example (II)

```
mod BAKERY is protecting NAT .
  sorts Mode State .
  ops sleep wait crit : -> Mode [ctor] .
  op <_,_,_,_> : Mode Nat Mode Nat -> State [ctor] .
  op initial : -> State .
  vars P Q : Mode .
  vars X Y : Nat .
  eq initial = < sleep, 0, sleep, 0 > .
  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y > if not (Y < X)
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .

  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
```

```
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y > if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
```

A Bakery Protocol Example (III)

We may wish to verify two basic properties about this protocol, namely:

- *mutual exclusion*, that is, the two processes are never simultaneously in their critical section, and
- *liveness*, that is, whenever a process enters the waiting mode, it will eventually enter its critical section.

Since the set of states reachable from `initial` is *infinite*, we should model check these properties using an abstraction. We can define the abstraction by adding to the equations of `BAKERY` a set E' of additional equations defining a quotient of the set of states. We can do so in the following module extending `BAKERY` by equations and leaving the rules unchanged:

A Bakery Protocol Example (IV)

```
mod ABSTRACT-BAKERY is
  inc BAKERY .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
  ceq < P, s X, Q, s Y > = < P, s s 0, Q, s 0 >
      if (Y < X) /\ not(Y == 0 and X == s 0) .
  ceq < P, s X, Q, s Y > = < P, s 0 , Q, s 0 >
      if not (Y < X) /\ not (Y == 0 and X == 0) .
endm
```

Three key questions are: (1) is the set of states now finite? (2) does this abstraction correspond to a rewrite theory whose equations are ground confluent, sort-decreasing and terminating? (3) are the

rules still ground coherent?

A Bakery Protocol Example (IV)

The equations are indeed ground confluent, sort-decreasing, and terminating. It is also clear that the set of states is now *finite*, since in the canonical forms obtained with these equations the natural numbers possible in the state can never be greater than $s(s(0))$.

Note (exercise) that the equivalence on states defined by the above equations is: $\langle P, N, Q, M\rangle \equiv \langle P', N', Q', M'\rangle$ iff:

- $P = P'$ and $Q = Q'$
- $N = 0$ iff $N' = 0$
- $M = 0$ iff $M' = 0$
- $M < N$ iff $M' < N'$

A Bakery Protocol Example (V)

This leaves us with the ground coherence question. We have to analyze possible "relative critical pairs" between rules and equations. Consider, for example, the following pair of a rule and an equation:

```
rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
```

The only possible overlap corresponds to the unification (after making the variables disjoint) of the two lefthand sides yielding the term, `< sleep,0,Q,s s Y >`, which is rewritten by the rule to, `< wait,s s s Y,Q,s s Y >`, and by the equation to, `< wait,s s 0,Q,s 0 >`, a term already in canonical form by the equations, to which the term `< wait,s s s Y,Q,s s Y >` is also rewritten using the conditional equation,

A Bakery Protocol Example (VI)

```
ceq < P, s X, Q, s Y > = < P, s s 0, Q, s 0 >
    if (Y < X) /\ not(Y == 0 and X == s 0) .
```

All the other rule-equation pairs can likewise be proved coherent. A fourth pending question is the `[State]`-*deadlock-freedom* of `BAKERY`, which fails to hold in general but holds for states reachable from `initial`. Since for reachable states this is not needed, we can leave things as they stand. Alternatively, we could use the general transformation described in **Ex**25.2. to obtain a `[State]`-deadlock-free version $\texttt{BAKERY}^{\texttt{[State]}}_{d.f.}$

What about *state predicates*? Are they preserved by the abstraction? In order to specify the desired mutual exclusion and liveness properties, it is enough to specify in Maude the following state predicates:

A Bakery Protocol Example (VI)

```
mod ABSTRACT-BAKERY-CHECK is inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .  inc ABSTRACT-BAKERY .
  ops 1wait 2wait 1crit 2crit : -> Prop .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < wait , X, Q, Y > |= 1wait = true .
  eq < P , X, wait, Y > |= 2wait = true .
  eq < crit , X, Q, Y > |= 1crit = true .
  eq < P , X, crit, Y > |= 2crit = true .
endm
```

Note that, since the rewrite theory is `State`-topmost and the equations are all on the kind of `State`, we only need to check that each of the equations preserves the above state predicates. But this is trivial, since the above predicates only depend on `Mode`

components that are left unchanged by the equations.

A Bakery Protocol Example (VI)

In other words, we have just shown that, for Π the above state predicates, we have a *strict* quotient simulation map,

$$\mathcal{K}(\texttt{BAKERY}, \texttt{[State]})_{\Pi} \longrightarrow \mathcal{K}(\texttt{ABSTRACT-BAKERY}, \texttt{[State]})_{\Pi}.$$

Therefore, we can establish the mutual exclusion property of `BAKERY` by model checking in `ABSTRACT-BAKERY-CHECK` the following:

```
reduce in ABSTRACT-BAKERY-CHECK : initial |= []~ (1crit /\ 2crit) .
result Bool: true
```

Likewise, we can establish the liveness property of `BAKERY` by model checking,

```
reduce in ABSTRACT-BAKERY-CHECK : initial |=
(1wait |-> 1crit) /\ (2wait |-> 2crit) .
```

```
result Bool: true
```

Exercises

Ex.25.1. Prove the following "universal property" of the quotient abstraction $q_{\equiv} : \mathcal{A} \longrightarrow \mathcal{A}/\equiv$ of a Kripke structure $\mathcal{A}$ on AP: For each AP-simulation map $h : \mathcal{A} \longrightarrow \mathcal{B}$ such that for each $a, a' \in A$, $a \equiv a' \;\Rightarrow\; h(a) = h(a')$, there is a *unique* AP-simulation map $\overline{h} : \mathcal{A}/\equiv \longrightarrow \mathcal{B}$ such that, $q_{\equiv}; \overline{h} = h$.

Ex.25.2. Given an *arbitrary* rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, whose equations E are ground confluent, sort-decreasing and terminating (with, say, Ω its subsignature of constructors) and whose rules R are ground coherent relative to E, define explicitly a protecting equational theory extension $(\Sigma, E) \subseteq (\Sigma', E')$, with the E' again ground confluent, sort-decreasing and terminating, where Σ' contains for each kind k a predicate $\mathit{enabled} : k \longrightarrow [\mathit{Bool}]$ such

that, for each $t \in T_\Sigma$ we have,

$$(\exists t')\ t \rightarrow^1_{\mathcal{R}} t' \quad \Leftrightarrow \quad enabled(t) = true.$$

Note: You do not need to give a full proof that the extension $(\Sigma, E) \subseteq (\Sigma', E')$, is protecting and has E' again ground confluent, sort-decreasing and terminating: a sketch of the argument and a correct construction are enough.

Ex.25.3. Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, whose equations E are ground confluent, sort-decreasing and terminating (with, say, Ω its subsignature of constructors) and whose rules R have "admissible" equational conditions (no rewrites in rule conditions, with the only extra variables (if any) in the condition beyond those in the lefthand side introduced by constructor patterns in "matching equations") are ground coherent relative to E, and with k a chosen kind of states, give an explicit construction of a theory extension $\mathcal{R} \subseteq \mathcal{R}^k_{d.f.}$, with the underlying signature and equations

of $\mathcal{R}^k_{d.f.}$, say, (Σ', E'), whose equational part is *protecting*, whose equations are ground confluent, sort-decreasing and terminating, and whose rules are ground coherent relative to the equations, and such that:

- $\mathcal{R}^k_{d.f.}$ is k'-*deadlock free* for a certain kind k', that is, for each $t \in T_{\Sigma',k'}$, $(\exists t' \in T_{\Sigma',k'})\ t \rightarrow^1_{\mathcal{R}^k_{d.f.}} t'$
- there is a function $h : T_{\Sigma',k'} \longrightarrow T_{\Sigma,k}$ inducing a bijection $h : T_{\Sigma'/E',k'} \longrightarrow T_{\Sigma/E,k}$ such that for each $t, t' \in T_{\Sigma',k'}$ we have,

$$h(t)(\rightarrow^1_{\mathcal{R}})^\bullet h(t') \quad \Leftrightarrow \quad t \rightarrow^1_{\mathcal{R}^k_{d.f.}} t'.$$

Furthermore, if Π are state predicates for $\mathcal{R}$ on the kind k defined by equations D, then one can define state predicates Π for $\mathcal{R}^k_{d.f.}$ on the kind k' by equations D' such that the above map h becomes a bijective bisimulation

$$h : \mathcal{K}(\mathcal{R}^k_{d.f.}, k')_\Pi \longrightarrow \mathcal{K}(\mathcal{R}, k)_\Pi$$

and therefore for any $\varphi \in LTL(\Sigma, E)$ we have,

$$T_{Reach(\mathcal{R})} \models_{LTL} \varphi \quad \Leftrightarrow T_{Reach(\mathcal{R}^k_{d.f.})} \models_{LTL} \varphi.$$

Note: You do not need to give a full proof that $\mathcal{R}^k_{d.f.}$'s equational part is protecting, its equations are ground confluent, sort-decreasing and terminating, and its rules are ground coherent relative to the equations: a sketch of the argument and a correct construction are enough.

Ex.25.4. Suppose a k-topmost rewrite theory in which all equations in E' are of the form,

$$t = t'$$

with t and t' of kind k. Suppose, furthermore, that the equations $E \cup E' \cup D$ are ground confluent, sort-decreasing, and terminating. Prove that if for each such equation and each state predicate p we

can prove the inductive property,

$$E \cup D \vdash_{ind} \ (\forall \vec{x}\, \forall \vec{y})\ (t(\vec{x})\,\texttt{|=}p(\vec{y}) = \texttt{true} \ \Leftrightarrow \ t'(\vec{x})\,\texttt{|=}p(\vec{y}) = \texttt{true})$$

then we have established the preservation of the state predicates Π by the equations E' in the precise sense that the implication,

$$[x]_{E \cup E'} = [y]_{E \cup E'} \quad \Rightarrow \quad L_\Pi([x]_E) = L_\Pi([y]_E)$$

holds.